INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

AN INTERPRETER FOR OBJECT COMPREHENSION QUERY LANGUAGE

Minh Hang Pham

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

November 1997

Abstract

An Interpreter for Object Comprehension Query Language

Minh Hang Pham

Object Comprehensions are a new query notation introduced in 1994 by Chan and Trinder to provide a declarative query language for object-oriented databases. Object Comprehensions allow queries to be expressed clearly, concisely, and processed efficiently, while incorporating many features that are missing from or inadequate in existing object-oriented query languages such as support of object-orientation, computational power and support of collection. However there is no object-oriented database (OOD) so far which incorporates Object Comprehension Language (OCL) as a query interface. This paper introduces an interpreter that evaluates the OCL query language against an in-memory database.

Acknowledgements

I would like to thank my supervisor, Dr. Gregory Butler, for his patience and valuable guidance.

I am also grateful to Georges Ayoub for the explanation of his collection utility routines, Bao Dang for his mentor during the source code writing process, and Darmalingum Muthiayen for his big help around the lab and particularly for his friendship which makes my school life much more enjoyable.

My special words of gratitude go to my family, especially my late father for his devotion thorough his life to provide the best for his children and my mother for her unceasing encouragement, support and sacrifice for her children's ambition.

Last, but not least, my appreciation goes to my better-half for all the sharing, love and understanding during my years in university.

Contents

List of Tables

List of Figures

Chapter 1

Background

1.1 Introduction

There are many object-oriented query languages [2, 7] that have been proposed in recent years. Some of them are designed particularly for object-oriented databases (OOD) and some are adapted from other areas: the relational data model and its extension [9], object-oriented programming languages [5]. According to [6], all of them are inadequate in one way or another. These inadequacies are categorised in [4] into four groups:

- *Support of object-orientation.* A few OO query languages do not capture the class hierarchy which is defined by the *ISA* relationship between classes defined in a database schema.

- *Structuring power.* It refers to the ability to explore and synthesize complex objects from the components of OOD. The creation of a new object may require a collection of objects as a parameter. To do so a query language must provide something like nested queries and allow orthogonal composition of constructs.

- *Computational power.* Recursion and quantification are two examples that characterise the computational power of a query language. Traversal of recursive queries as well as quantification are supported poorly. Recursive queries with computation are supported even worse.

- *Support of collection.* Usually "Set" is widely supported and its operations are well defined. It is not the case for other collection classes. Interaction between

the different collection classes is also unclear.

To overcome the above-mentioned inadequacies, Chandra and Trinder [4] introduces a new query notation called *Object Comprehensions*, which takes into consideration the fundamental properties of object-oriented data models and eliminates the drawbacks above. Their Object Comprehensions Language (OCL) is clear, consise and powerful. The extension of List Comprehensions [10] to Object Comprehensions was done by consolidating and improving constructs found in existing query languages.

1.1.1 Overview of the OCL Project Components

The interpreter in this report is responsible for evaluating the OCL query on a database which resides in the memory. The database setup as well as update, and search for the proper tuple/record is handled by the DBEntity class. DBEntity interacts directly with the database and in the meantime, is the bridge between the interpreter and the database. The Database DBEntity classes are created by Dr. Gregory Butler. The Database class handles the creation of the database, and the search. It returns DBEntity objects, that are responsible for updates, and navigation. In parallel with the interpreter presented in this paper, there are two other projects that are involved with OCL. The first project is the creation of an Ode-based database. which is used as a target database for the second one, the translation of the OCL queries into the O++ query language that is used by Ode. The author of the first project is Georges Ayoub, and of the second one is Alexander Lakher. Even though this interpreter does not use the Ode-based database, the above three projects are interrelated. The implementation of the interpreter employs the utility collection routines created by G. Ayoub. Furthermore, the understanding of the OCL query notation is enhanced thanks to the report of A. Lakher.

1.1.2 Organization

The organisation of this report is as follows. The remaining sections of this chapter provide a background of Comprehensions and an overview of the interpreter. Chapter 2 describes the sample data model, OCL and the sample queries. Chapter 3 outlines the implementation of the interpreter. Chapter 4 concludes. The bibliography is followed by appendices which contain the actual code of the interpreter.

1.2 Comprehensions

1.2.1 Set Comprehensions

The inspiration for *comprehensions* was the standard and convenient mathematical notations for sets. For example in mathematics the set of squares of all the even numbers in a set S would be written as:

$$\{x^2 | x \in S \wedge even(x)\}$$

Comprehensions first appeared as *Set Comprehensions* in an early version of the programming language NPL that later evolved into *Hope* [12] but without comprehensions. They were followed by *List Comprehensions* [13].

1.2.2 List Comprehensions

A full description of *List Comprehensions* can be found in [10]. The above mathematical expression written using *List Comprehensions* would have a look:

$$\{x^2 | x \leftarrow L; even(x)\}$$

where L stands for a list.

List Comprehensions have been incorporated into several functional languages, e.g. Miranda [14] and Haskell [8]. The syntax of list comprehensions is:

$$E ::= \text{"[" } E \text{ "|" } Q \text{ "]"}$$

$$Q ::= E \mid P \leftarrow E \mid \varepsilon \mid Q \,;\, Q$$

where E stands for an expression, Q stands for a qualifier, P stands for a pattern, and ε stands for an empty qualifier.

The result of evaluating the comprehension [E | Q] is a new list, computed from one or more existing lists. The elements of the new list are determined by repeatedly evaluating E, as controlled by the qualifier Q. A qualifier is either a filter or a generator, or a sequence of these. A filter is a boolean-valued expression, expressing a condition that must be satisfied in order for an element to be included in the result.

An example of a filter in the example above is *even(x)*. A generator of the form T $\leftarrow$ E, where E is a list-valued expression, makes the variable T range over the elements of the list. An example of a generator is $x \leftarrow L$ above. More generally, a generator of the form P $\leftarrow$ E contains a pattern P that binds one or more new variables to components of each element of the list.

1.2.3 Object Comprehensions

A generalisation of list comprehensions is *collection comprehensions*, which provide a uniform and extensible notation for expressing and optimizing queries over many collection classes including sets, lists, bags, trees and ordered sets. That is what Chandra and Trinder [4] have called *object comprehensions*. The most significant benefit is that, although each primitive operation will require a separate definition for each collection class, only one query notation is needed for all these collection classes. A single definition is all that is required for high-level operations defined in terms of collection comprehensions. It significantly reduces the syntactic complexity of the query notation:

$$\{x^2 | x \leftarrow C; even(x)\}$$

where C stands for a collection, i.e. could be a bag, a set or a list. Here are some examples of object comprehensions using the notation suggested in [4].

Example 1. Return a set of elements of the bag B. A bag allows duplicates, however they are to be eliminated in the resulting set.

```
Set [ e <- B | e ]
```

Example 2. Return a bag of elements of the list L. Possible duplicates are preserved, however the order of the elements is lost.

```
Bag [ e <- L | e ]
```

Example 3. Return a list of elements of the list L provided they comply with some specific 'condition'.

```
List [ e <- L ; condition(e) | e ]
```

It is worth mentioning that comprehensions are a declarative specification of a query, and as it is shown in [4], are a good query notation for being concise, clear, expressive

and easily optimized. However the optimization was out of scope of this project and left for the future work.

1.3 Overview of the Interpreter

The interpreter for the Object Comprehension Query is based on a BNF (Backus Naur Form) grammar, the OCL grammar itself is presented in Chapter 2. From the grammar, we can automatically construct an efficient parser that determines if a query is syntactically well formed. The interpreter is structured as a typical compiler. The parser, which obtains a string of tokens from the lexical analyzer, verifies that the string can be generated by the grammar and constructs a parse tree for query evaluation as indicated in Figure 1.

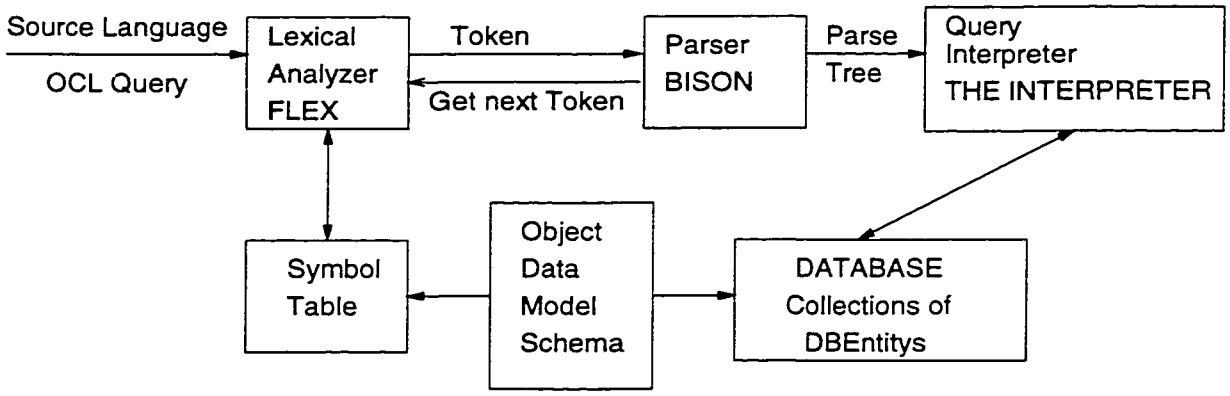

Figure 1: Interpretation Process for OCL Query Language.

The parsing technique using for this interpreter is LR(1) parsing, which scans the input from left to right, conducts a rightmost derivation in reverse and uses one input symbol of lookahead to make a parsing decision. BISON [11], which is similar to YACC, is used to create the parser. It uses the scanner generated by FLEX [15]. BISON generates an integrated parser, a file of C program, which provides for semantic stack manipulation and the specification of semantic routines. BISON is used instead of YACC since it is more compatible with C++. The query evaluation process is then implemented in C++.

Chapter 2

OCL

The sample data model is a simplified university system which records information about students, staff members of a university, its academic departments and courses. Figure 2 presents the data model.

2.1 The Sample Data Model

The class *Person* has two subclasses: *Student* and *Staff*; *Visiting Staff* is a subclass of *Staff*. *Tutor* inherits from both *Student* and *Staff* to incorporate students doing part-time teaching. The calculation of the salary of a tutor is different from that of a staff member. The variation is captured by overloading *Salary* method to *Tutor*. Every person has an address which is an object of the class *Address*. A student has at least one superviser. This is modelled by the *SupervisedBy* method as a list of staff members. Every and each student as well as a staff member is associated to a department of class *Department* by means of *Major* and *department* accordingly. Courses of class *Course* are *runBy* a department provided there is a staff member who *teaches* this particular course which enables a student to take this course via *takes*. A course may have a set of *prerequisites*. The following table presents the schema definition. Using this model the next section describes OCL by presenting different kinds of queries in OCL notation.

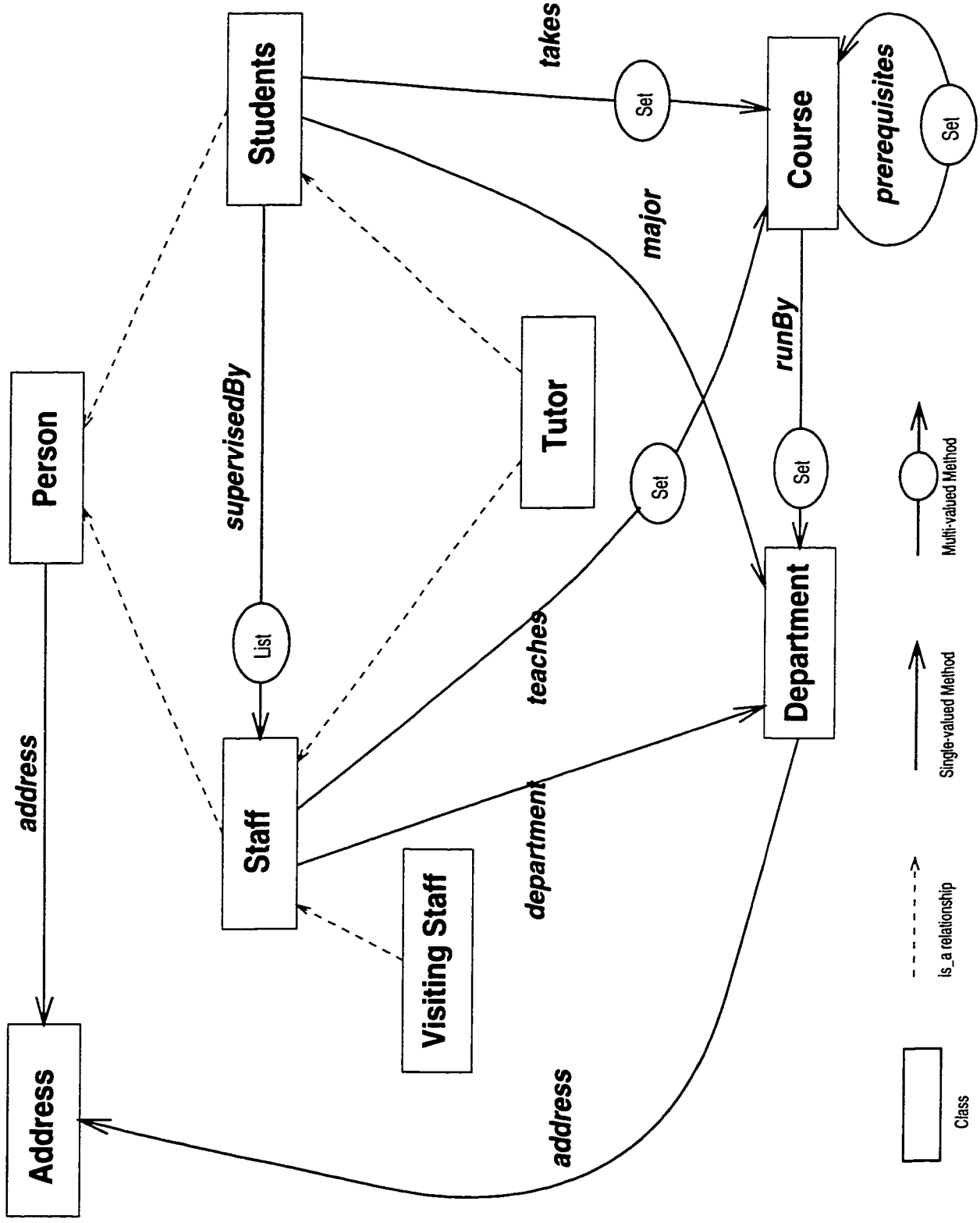

Figure 2: University Model Diagram.

Table 1: The Schema Definition.

```
Class Person isa  Entity
 methods
 name         : -->  String ,
 address      : -->  Address.

Class Staff isa  Person
 methods
 department   : -->  Department,
 teaches      : -->  Set of Course,
 salary       : -->  Integer.

Class Student isa  Person
 methods
 major         : -->  Department,
 takes         : -->  Set of Course,
 supervisedBy  : -->  List of Staff.

Class Tutor isa  Student, Staff
 methods
 salary        : -->  Integer.

Class Course isa  Entity
 method
 code          : -->  String,
 runBy         : -->  Set of Department,
 prerequisites: -->  Set of Course,
 assessments   : -->  Bag of Integer,
 credits       : -->  Integer.

Class VisitingStaff isa  Staff.
Class Address isa  Entity
 methods
 street        : -->  String,
 city          : -->  String.

Database is
  Persons      :  Set of Person,
  Departments  :  Set of Department,
  Courses      :  Set of Course,
  StaffMembers:  Set of Staff,
  Students     :  Set of Student,
  Tutors       :  Set of Tutor.
```

2.2 The OCL Sample Queries

Before starting the query examples we would like to elaborate on the grammar of the object comprehensions. Table 2 presents the OCL Grammar. That would be useful while examining the queries and also later when discussing the implementation issues of the interpreter in Chapter 3.

Table 2: The OCL Grammar.

```
expr_list       ::= expr | expr, expr_list
expr            ::= expr "union" expr | expr "differ" expr
                  | expr "and" expr | expr "or" expr
                  | "not" expr | identifier [ "." expr ]
                  | identifier "(" expr_list ")" | "size" expr
                  | constant | expr "hasclass" expr
                  | expr "hasclass" expr "with" expr
                  | expr "(" expr ")"  | expr arith_op expr
                  | quantifier expr op Quantifier expr
                  | collection_kind "{" expr_list "}"
                  | collection_kind "{" expr "..." expr "}"
                  | {collection_kind "[" qualifiers "|" expr "]"}
collection_kind::= "Set" | "Bag" | "List"
qualifiers      ::= qualifier [ qualifiers ]
qualifier       ::= [ generator ] | [ localdef ] | [ expr ]
generator       ::= identifier "<-" expr
localdef        ::= identifier "AS" expr
quantifier      ::= [ some of ] | [ atleast expr of ] | [ just expr of ]
                    | [ atmost expr of ] | [ every of ]
op              ::= ">" | "<" | "<=" | ">=" | "==" | "!="
arith_op        ::= "*" | "/" | "+" | "-"
```

Presented in conventional BNF it still might need some explanation. The best way is to do it by examples. The example of a typical *expr* that can be used in a localdef is a compound indentifier `s.address.city`. Another type of *expr* that can be used in a query is `s.address.city == "Vienna"`. A *generator* example: `s` ← `Students`. A *localdef* example: `a AS s.address.street`. A query then would be expressed in the following style:

```
Set [ s <- Students ; a AS s.address.city ; a == "Vienna" | s ]
```

2.2.1 OCL Sample Queries

The following novel features of OCL as a query language should be noted:

- a predicate-based optimizable language providing support for the class hierarchy;
- numerical quantifiers for dealing with occurrences of collection elements;
- operations addressing collection elements by position and order;
- a high-level support for interaction between different collection kinds;
- recursive queries with computation.

Now we are to demonstrate object comprehensions by using queries posed against the described university database. Queries Q1 - Q6 demonstrate the support of Object-Orientation; Q7 and Q8 explore the result expression. Q9 - Q11 focus on generators. Quantifiers are highlighted in Q12 - Q17. Support of Collection is emphasized in Q18 - Q27. The last query, Q28, is to reflect the ability of Query Functions and Recursion. All queries demonstrate the declarative nature of object comprehensions as a query notation.
Each query presented in the following format:

Query #. The text of a query

OCL version of a query

Method Calling and Dynamic Binding.
Encapsulation protects attributes of an object from being accessed directly. An access is to be made via a method. In Q1 *s.salary* represents the calling of method *salary* on a staff member object *s* drawn from StaffMembers. A method could be overloaded

as it as the case with *tutor* whose salary is calculated in a different way.

Q1. Return staff members earning more than $1000 a month.

```
Set [ s <- StaffMembers; s.salary > 1000 | s ]
```

Complex Objects and Path Expressions.
Support of complex objects implies that a method call may return an object which can, in turn, receive another method call and so on. Such a sequence of method calls is usually referred to as a path expression.

Q2. Return tutors living in Glasgow.

```
Set [ t <- Tutors; t.address.city = "Glasgow" | t ]
```

Object Identity.
In object-oriented data models, objects are represented by object identifiers which are essential for object sharing and representing cyclic relationships. Equality between objects is defined by the equality between their object identifiers.

Q3. Return tutors working and studying in the same department.

```
Set [ t <- Tutors; t.department = t.major | t ]
```

Class Hierarchy.
StaffMembers contains only members of the faculty. The only collection in the database that contains all visiting staff members is *Persons*. The elements of *Persons* can be of class *Persons* or its subclasses. In Q4 *hastype* returns true if person object p is an instance of class *VisitingStaff*.

Q4. Return all visiting staff members in the university.

```
Set [ p <- Persons; p HASTYPE VisitingStaff | p ]
```

In the following query the method *salary* is defined for visiting staff members but not for persons in general. The filter is applied only if the object is of specified class.

Q5. Return visiting staff members earning more than 1000 a month.

```
Set [ p <- Persons; p HASTYPE VisitingStaff WITH p.salary > 1000 | p ]
```

Local Definitions.

Local definitions simplify queries by providing symbolic names to expressions. They are particularly useful when a path expression is used in more than one place.

Q6. Return students whose major departments are in either Hill st. or U ave.

```
Set [ s <- Students; a AS s.major.address.street; a == "Hillst"
        OR  a == "Uave" | s ]
```

The Result Expression.

Operation that create new objects should be allowed in the result expression. Method *new* takes to parameters and creates a new object of class *AClass* for each object in *Students*. In the next query the result is obtained by creating new objects using the student objects and the sets of courses.

Q7. Return students and courses taken by them.

```
Set [ s <- Students | ACLASS.NEW(s, s.takes) ]
```

Nested Queries.

The result is obtained by creating new objects using the student objects and the sets of courses. Nested queries enable richer data structures to be returned and complex selection conditions to be expressed. An inner query above is a parameter to the method call in the result expression of the outer query.

Q8. Return students and courses taken by them with a credit rating over one.

```
Set [ s <- Students | ACLASS.NEW(s, Set[c<-s.takes;c.credits>1|c])]
```

Multiple Generators.

Multiple generators allow relationships that are not explicitly defined in the database schema to be reconstructed. Two variables can range over the same set independently.

Q9. Return students studying in the same department as SteveJ.

```
Set [ x <- Students; y <- Students; x.name == "SteveJ";
                    x.major == y.major | y]
```

Dependent Generators.

Dependent generator is used to facilitate querying over the elements in a nested collection.

Q10. Return courses taken by the students.

```
Set [ s <- Students; c <- s.takes | c ]
```

Literal Generators.

Collection literals can simplify queries by making them more concise and clearer.

Q11. Return those courses among C1,C2,C3 which have a credit rating over one.

```
Set [ c <- Courses; x <- Set ["C1","C2","C3"]; c.code == x ;
                    c.credits > 1 | c ]
```

Existential Quantifiers.

A restricted form of existential quantification is provided by *some*, which can appear on either side of an operator. In Q12, the filter succeeds if course code is one of the memberslisted. In Q13, the filter returns true if there is a common element between the two sets; i.e. an non-empty intersection.

Q12. Return those courses among C1,C2,C3 which have a credit rating over one.

```
Set [ c <- Courses; c.credits > 1;
          c.code = SOME Set ["C1", "C2", "C3"] | c]
```

Q13. Return students taking a course given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
          s <- Students; SOME s.takes = SOME j.teaches | s]
```

Universal Quantifiers.

In the following query the keyword *EVERY* is the universal quantifier. The filter succeeds if all the course elements in *s.takes* are also in the set *j.teaches*; i.e. subset.

Q14. Return students taking only courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
      s <- Students; EVERY s.takes = SOME j.teaches | s]
```

Numerical Quantifiers.

In the following queries the keywords *ATLEAST*, *JUST* and *ATMOST* are the numerical quantifiers. Numerical quantifiers are very useful in dealing with duplicate elements in collections and the number of elements that are common between two collections, i.e. the size of intersection. In Q15, the filter turns true if there are two or more elements that are common between specified sets. In Q16, the filter succeeds if there are exactly two common elements. In Q17, the size of intersection must be less than or equal to two.

Q15. Return students taking two or more courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
         s <- Students; SOME s.takes = ATLEAST 2 j.teaches | s]
```

Q16. Return students taking exactly two courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
         s <- Students; SOME s.takes = JUST 2 j.teaches | s]
```

Q17. Return students taking no more than two courses given by Steve Johnson.

```
Set [ j <- StaffMembers; j.name = "SteveJohnson";
         s <- Students; SOME s.takes = ATMOST 2 j.teaches | s]
```

Aggregate Functions.

The aggregate function *size* returns the number of elements in a collection. It is defined for all collection classes. For bags and lists duplicate elements are included in the counting. Some aggregate functions are possibly defined only for certain collection classes.

Q18. Return courses with less than two assessments.

```
Set [ c <- Courses; c.assessments.size < 2 | c ]
```

Equality.

It is quite a necessity to have an ability to compare two collections based on the elements, occurences and their order. Thus two bags are equal if for each element drawn from either collection there is equal number of occurences in both bags. For the lists, number of occurences and the positions must be the same.

Q19. Return courses requiring no prerequisite courses.

```
Set [ c <- Courses ; c.prerequisites == Set [ ] | c ]
```

Occurrences and Counting.

Bags and lists allow duplicates. The following two queries are to show how the occurences of elements could be used.

Q20. Return courses with four 25% assessments.

```
Set [ c <- Courses; JUST 4 c.assessments = 25 | c ]
```

Q21. Return the number of assessments counted 25% in the course "db4".

```
Set [ i <- List{0..db4.assessments.size };
                        JUST i db4.assessments = 25 | i]
```

Positioning and Ordering.

A list allows duplicates and keeps track of the order of the elements. In Q22, the first two elements of the list are returned and used in a generator. In Q23, a sublist whose first element is Steve and whose last element is Bob is returned. It returns an empty list if Steve does not come before Bob in a supervisor list.

Q22. Return the first and the second supervisors of Steve Johnson.

```
Set [ s <- Students; s.name = "SteveJohnson";
      p <- s.supervisedBy.[1..2] | p ]
```

Q23. Return students having Steve before Bob in their supervisor lists.

```
Set [ s <- Students; s.supervisedBy.[Steve : Bob] ~== List [ ] | s]
```

Union.

The *union* operator combines two collections to form a new collection of the same class but having all the elements. The union of bags contains all the elements including the duplicates. The union of a list to another one appends the latter one to the former one.

Q24. Return students in the CS and EE departments.

```
Set [s <- Students; s.major.name = "CS" | s ]
    UNION Set [s <- Students; s.major.name = "EE" | s ]
```

Differ.

For the *differ*, the class of the result elements is determined in the same way as in union. The number of occurences for an element in the result collection is the difference of that in the operand collections. For lists, differ will remove the last match.

Q25. Return cities where students, but no staff, live.

```
Set [s <- Students | s.address.city ]
    DIFFER Set [s <- StaffMembers | s.address.city ]
```

Converting Collections.

There might be a need to convert a bag, a list or a set one to another. Bag to set conversion would eliminate duplicates while converting to a list would involve an additional effect of assigning an arbitrary order over the result elements.

Q26. Return the salary of tutors and keep the possible duplicate values.

```
Bag [ t <- Tutors | t.salary ]
```

Mixing Collections.

Object-oriented data model supports more than one kind of collection. Hence the corresponding query notation should support not only different collection classes but also the mix of them in the same query. In the following query, *s.supervisedBy* returns a list and is mixed with two generators drawing from sets. It should be mentioned that swapping of generators will not be allowable if the result collection is to be a list.

Q27. Return courses taught by the supervisors of Steve Johnson.

```
Set [ s <- Students; s.name = "SteveJohnson";
          sup <- s.supervisedBy; c <- sup.teaches | c]
```

Query Functions and Recursion.

It is natural to find the cyclic relationships in object-oriented data models. This implies recursion support. The recursive queries can be expressed in object comprehensions via query functions. In the following query the result is generated by retrieving elements from a collection returned by a recursive function, *f(c.prerequisites)*. This function takes a set of courses and returns a set of courses. For each element drawn from the input collection, f is applied recursively on the prerequisite courses, and the result is then used as a part of the input. The recursion terminates when f is passed an empty set.

Q28. Return all direct and indirect prerequisite courses for the "DB4" course.

```
let f(cs : Set of Course ) be
        cs UNION Set [x <- cs; y <- f(x.prerequisites) | y]
    in Set[c <- Courses; c.code = "DB4"; p <- f(c.prerequisites) | p]
```

Chapter 3

Interpreter

This chapter discusses the implementation of the interpreter, the interpreter code is provided in appendix A (*.h files) and appendix B (*.C files), and the grammar is in appendix D. The interpreter takes as input a program written in OCL, analyzes it to identify tokens then carries out the actual evaluation on the database.
The issues of translator, interpreter and compiler design are well addressed in [1, 3].

3.1 The Phases of Interpretation

The implementation of the interpreter is divided into phases with the output of each phase passed as input to the next phase.

3.1.1 Lexical Analyzer

The first phase of the interpreter is scanning each statement of the source language and recognizing the tokens specified by the regular expression. This process of specifying token patterns is done by the *lexical analyzer*. *FLEX* is used to produce a *lexical analyzer* that can be used with *BISON*. The *FLEX* library *ll* will provide a driver program named *yylex()*, the name required by *BISON* for its *lexical analyzer*.
Based on the grammar shown in chapter 1, a token is one of the following:

- Valid operators such as , ; + * ==, etc.
- Valid keywords such as union, differ, atleast, etc.
- Constant token consisting of digits; its value is stored in yylval.innumb.

- Identifier token consisting of strings; its value is stored in yylval.instring.
- Literal token consisting of characters in quotes; its value is stored in yylval.instring as well.

These specifications are stored in *tokendef.l* file. *FLEX* then produces a stream of tokens and passes them as input to the next phase, the *syntax analyzer* or *parser*. The following example is used to elaborate the definition of different types of tokens.

Example 1

Input:

```
Set [ s <- Students ; a AS s.major.address.street ;
      a == "Hillst" ; a == "Uave" | s ]
```

Output:

```
token   < Set >
token   < LFSQBKTK >
token   < identifier  s >
token   < LIMPLTK >
token   < identifier  Students >
token   < SEMICOLONTK >
token   < identifier  a >
token   < ASTK >
token   < identifier  s >
token   < DOTTK >
token   < identifier  major >
token   < DOTTK >
token   < identifier  address >
token   < DOTTK >
token   < identifier  street >
token   < SEMICOLONTK >
token   < identifier  a >
token   < EQTK >
token   < literal  "Hillst" >
token   < SEMICOLONTK >
```

```
token   < identifier  a >
token   < EQTK >
token   < literal  "Uave" >
token   < BARTK >
token   < identifier  s >
token   < RSQBKTK >
```

3.1.2 Parser

BISON is responsible for generating the parser. The *Parser* takes as input the stream of tokens produced by the *Lexical analyzer* which is mentioned in the previous phase. The *grammar* that will be processed by the parser-generator *BISON* to produce a parser is contained in the file *grammar.y*. File *grammar.y* is shown in Appendix D and has three main sections:

- A list of tokens or terminal symbols. In the following example, IDENTIFIERTK and CONSTANTTK are terminal symbols, and the rest are tokens.

```
%token <instring>        COMMATK
%token <instring>        SEMICOLONTK
%token <instring>        PLUSTK
%token <instring>        MINUSTK
%token <instring>        DIVTK
%token <instring>        IDENTIFIERTK
%token <innumb>          CONSTANTTK
```

- A list of variables, subprogram headers, the name of the start symbol, and the declaration of the precedence and associativity to resolve the ambiguity of the grammar. A sample of this section with the list of variables and the precedence and associativity relationship between tokens is presented below:

```
%union
{
        char            instring[64];
        int             innumb;
```

```
        int                 CollectionType;
        Prog                *prog_ptr;
        DList≺Expr≻         *expr_list_ptr;
        Expr                *expr_ptr;
        Qualifier           *qual_ptr;
        DList≺Qualifier≻    *qual_list_ptr;
        Generator           *generator_ptr;
        Localdef            *localdef_ptr;
        Hasclassqual        *hasclass_ptr;
        Hasclasswithqual    *hasclasswith_ptr;
}

%nonassoc    THREEDOTTK
%left        MINUSTK PLUSTK
%left        MULTTK  DIVTK
%nonassoc    UMINUS
%left        DOTTK
```

- A productions section to define the *grammar*. Productions are of the form *A:B1...Bn*, where *A* is the left hand side of the production, and *B1...Bn* are zero or more terminal or non-terminal symbols. Example 2 below will illustrate this concept.

Example 2

```
expr_list:
        expr
        {
        $$ = new DList ≺ Expr ≻;
        $$ → Insert($1);
        } |
        expr COMMATK expr_list
        {
```

```
                $$ = $3;
                $$ → Insert($1);
                }
;
```

In the above production, *expr_list* can be parsed in two ways: it can either become an *expr* or become an "*expr COMMATK expr_list*". As we can see, the left hand side contains one non-terminal symbol *expr_list*; the right hand side contains one non-terminal symbol *expr* for the first option of the production, and two non-terminal symbols: *expr* and *expr_list* and one terminal symbol *COMMATK* for the second option of the production.

To generate code for each of the productions, a semantic routine, written in C++, is inserted on the right hand side. These semantic routines are treated as action symbols for *BISON*, and may access and update the semantic stacks of *BISON*. As in Example 2 above,

```
$$ = new DList ≺ Expr ≻;
$$ → Insert($1);
```

are semantic routines written in C++, incorporated with *BISON* notation, to parse *expr_list* into *expr*. Similarly, the semantic routines used to parse *expr_list* into "*expr COMMATK expr_list*" are:

```
$$ = $3;
$$ → Insert($1);
```

As can be seen from the *grammar* in Appendix D, different types of expressions are created on different production rules depending on the terminal symbol called *op-code* which can be found on the right hand side. When a query statement is read, the *Expr_query* object of type *expression* is created. It consists of a token *SETTK*, *LISTTK* or *BAGTK*, a list of *qualifiers*, a token *BARTK* and a return object of type *expression*. As we can see, the list of *qualifier* is the heart of the query statement. Please see the complete declarations of *qualifiers* and *expressions* in *expr.h* and *qual.h* in Appendix A for details.

There are three main types of *qualifier*: *generator*, *local definition*, and *expression*. *BISON* is responsible for handling the differentiation among the *qualifiers* and creates them accordingly.
The output of a *parser* is a *Parse Tree*, which is a syntactic structure that incorporates all recognized structures. The complete hierarchy of a *qualifier* is shown in Figure 3. Let us continue the previous example, assuming that the stream of tokens was fed into the parser. Thus the input is the output shown in the Example 1. Figure 4 presents the object model of the query in Example 1 that the parser produces.

3.1.3 Interpreter

Once the qualifier is properly recognized, the *interpretation process* takes place. It is implemented in C++ to take the full advantage of the class hierarchy and polymorphism characteristic of C++. The main *interpretation process* is handled by the virtual method *evaluate()*. An *OCLVALUE* class hierarchy is created to support different kinds of intermediate results during the *evaluation process* of the *parse tree*. The final result of the query statement is stored in an *OCLVALUE* object as well.
Another important function in the *interpretation process* is to ensure that the left and the right hand side of the expression have the same type. This is handled by the virtual method *typechecking()*, which is also responsible for storing the information read from the parser in the proper *bookeeping table* depending on the expression type. Details of the two principal functions *typechecking()* and *evaluate()* for each particular expression type will be discussed below.

Expression

Here we discuss the expression of type *Expr*. *Expr* is a basic structure or a smallest unit that the user can employ in a query statement to express the criteria and constraints to search the database. Thus, it will be discussed in more detail than any other type. Figure 5 presents the basic components of an *Expr* object.

An *Expr* object has a pointer to *Leftoperand*, a pointer to *Rightoperand*, and an *op*. *Leftoperand* and *Rightoperand* are again of type *Expr*. Any pointer can be null, as in the case of an *Unary Expr*, in which case the pointer to *Leftoperand* is null. *Op* is a short name for operator. It defines the relationship between the *LeftOperand* and the

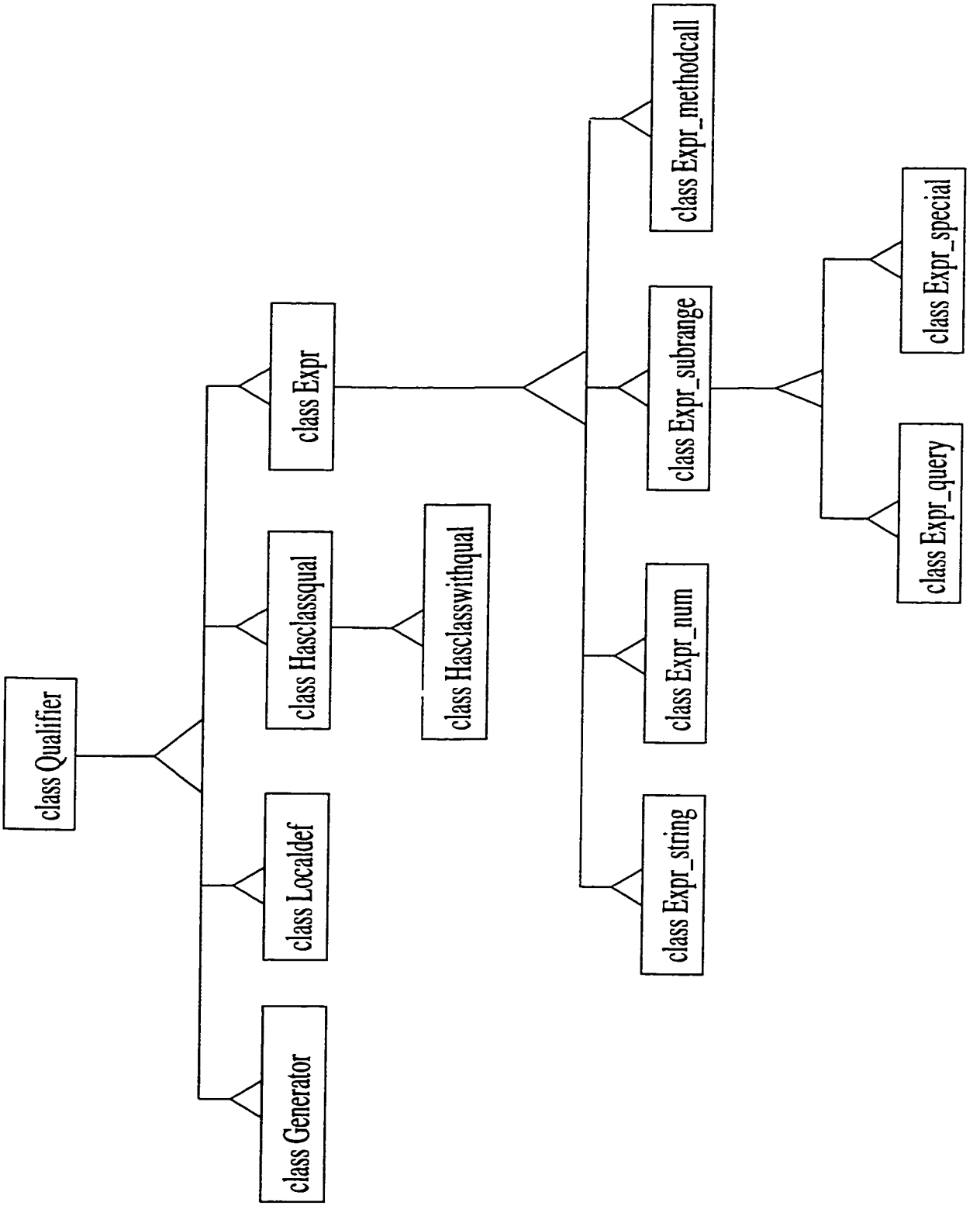

Figure 3: Qualifier Hierarchy.

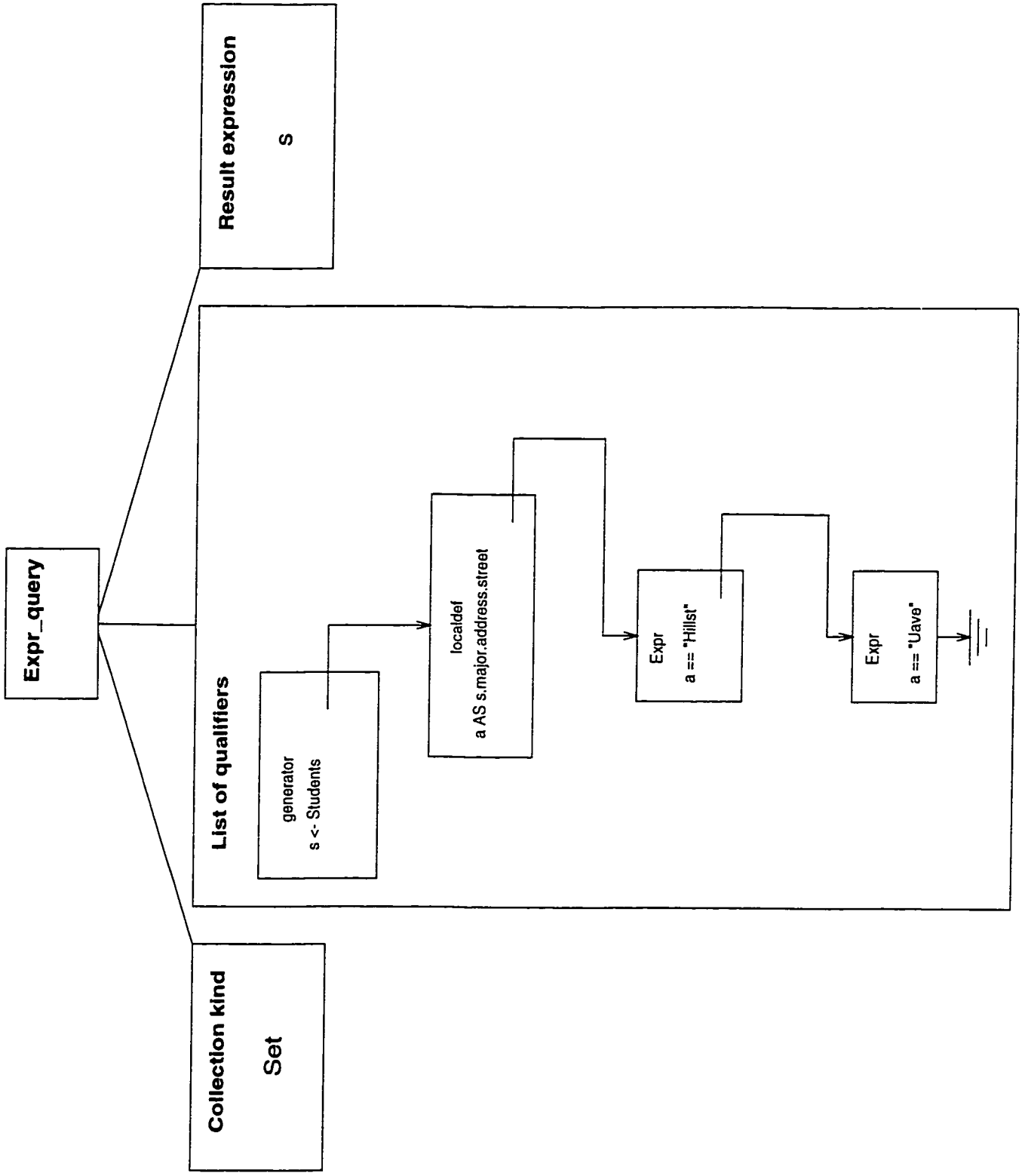

Figure 4: Example 3: The Object Model for Example 1 after Parsing.

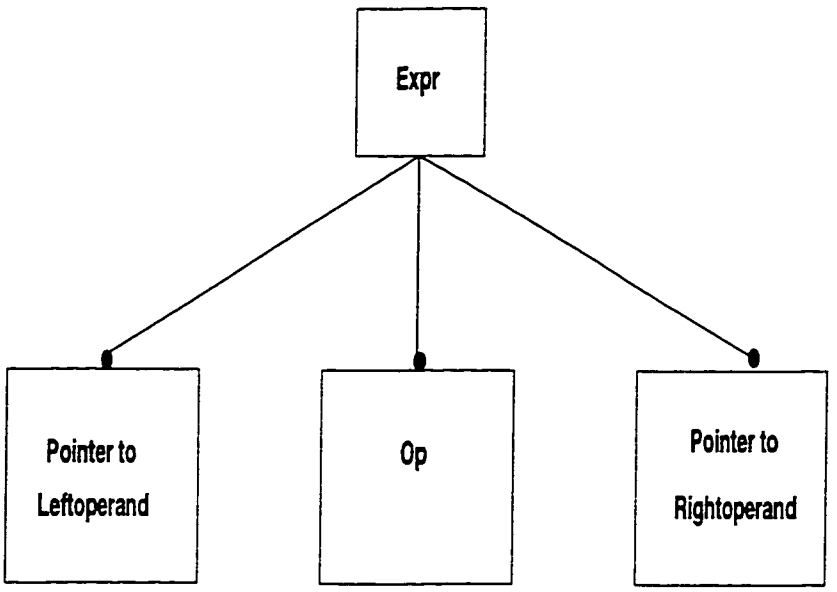

Figure 5: The Object Diagram of Expr

RightOperand. It is of type Opcode which is an enumerated type. Figure 6 describes the main methods of class *Expr*. The first three methods *get_leftop()*, *get_rightop()*, and *getop()* are the basic operations to access the three components *Leftoperand*, *Rightoperand*, and *op* of a particular *Expr* object. The two methods *typechecking()* and *evaluate()* are defined virtual. They are the two most important methods that are redefined at any subclass of *qualifier*. The functionality of *typechecking()* is to ensure that the type of *Leftoperand* matches the type of *Rightoperand*, and *evaluate()* does nothing other than evaluating the *Expr*. Both *typechecking()* and *evaluate()* methods are done in a bottom up fashion in the *Expr parse tree*. Each starts at the leaf level of the *parse tree*, where its left child is checked and evaluated before its right child.

Type check the expression *typechecking()* is responsible for:

- Ensuring the type of *Leftoperand* matches the type of *Rightoperand*. It also validates that type against the types allowed by *op*;
- Returning the type of *Expr* if the above condition is true, otherwise it returns *NULL*.

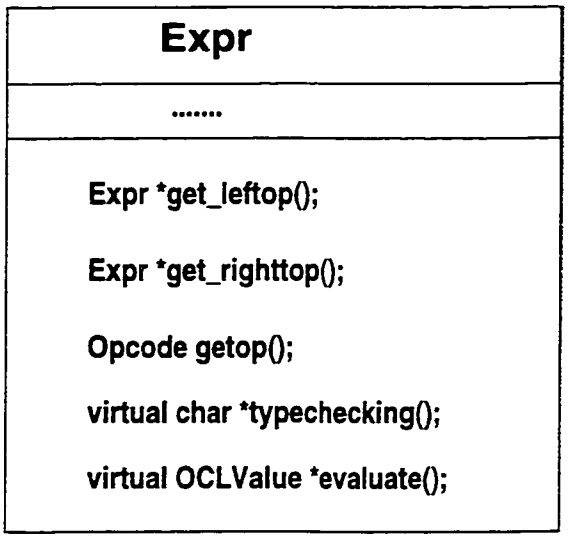

Figure 6: The Methods of Expr

Evaluate the expression The actual operation of the opcode is carried on at this stage. *Exp* is evaluated in a bottom up fashion as mentioned above. The result of *evaluate()* is an *OCLVALUE* object. The description of the *OCLVALUE* class is discussed in section 3.1.5.

Generator

Generator is a means to introduce a new variable and its type to be used in a query statement.

Type check the generator expression *Typechecking()* ensures that the identifier on the left hand side does not already exist. If it exists, *typechecking()* will print out an error message and return NULL. If it does not, *typechecking()* adds a new record in the *variable bookeeping table*. This record contains the variable name, which is the identifier itself, type of the variable, which can be found on the right hand side of the *Generator*, and a pointer to the database storage which contains such objects. *Typechecking()* returns the type of the identifier read on the left hand side of the *Generator* or *NULL* if it's invalid.

Evaluate the generator expression *Evaluate a generator* will return an object from the database. On each activation of *evaluate()*, an object of the proper type

is fetched from the database and the pointer is updated to the next object until the database is exhausted.

Local Definition

Local Definition is also a means to introduce a new variable in a query statement. The type and the value of the new variable is determined by expanding the expression following the keyword *AS*.

Type check the local definition expression *Typechecking()* at first ensures that the new variable introduced, the identifier preceding *AS*, has not been defined elsewhere in the query statement. It then stores the new variable in a *macro bookeeping table*, with the same type as the expression following *AS*, and a pointer to that expression as well.

Evaluate the local definition expression For the *local definition expression*, *evaluate()* always returns an intermediate *OCLBOOL* with true value. The reason for this is the expression following *AS* gets *expanded* or *evaluated* only when the variable preceding *AS* is referred again in the query statement.

3.1.4 Table-Management and Error Handling

All of the above phases of the *interpreter* are involved with the *bookkeeping* or *table-management* activity to keep track of the variable names used by the query statement and to record their type information. In this section, only the tables created at the interpretation process will be discussed in details. Others at the *lexical analyzer* and *parser* phases are created and handled by *FLEX* and *BISON* respectively.
The *interpreter* has two main tables: one to maintain the necessary information of the variable or identifier read from the query statement, such as name, type and its current object. The other table is to store name and the macro to be expanded for the *local definition expression*. They are both organized into hash tables, since this is generally the preferred and accepted method of handling symbol tables as stated in [1] due to the efficiency reasons. An object of class *hashTable* handles the bookeeping functionality for regular variables, and an object of class *funcTable* handles the bookeeping functionality for macro variables. *Hash coding* is used to organize both

tables. It uses some computable function of the numeric representation of the name to determine at which point in the list the name should be entered. This is accomplished by the method *hash* of both classes *hashTable* and *funcTable*. Files *table.h* and *table.C* contain the code. *Error Handling* is another activity of all phases. The error handler is to be invoked whenever a flaw in the source is detected. Then a warning error message is to be issued. There is no error handler in our interpreter at this time. Error handling is done by printing a message by the *typecheck()* routine that discovers a flaw. Again, both table-management and error handling interact with all phases.

3.1.5 Intermediate Values - OCLVALUE

The *OCLVALUE* class is created to handle the intermediate values as well as the final result of the *evaluate()* phase of the *interpreter*. It has four main subclasses: *OCLNumber*, *OCLBoolean*, *OCLString*, and *OCLCollection*. As the name suggests, *OCLNumber* handles the arithmetic operations, *OCLBoolean* handles the logical operations, *OCLString* handles the string operations, and *OCLCollection* handles the collection operations. Again, the result of each of the above operations is stored in an object which belongs to one of the above classes depending to the nature of the result.

3.1.6 Passes

There are *multi-pass* and *single-pass* interpreters. A pass reads the source, which can be the output of the previous pass, makes the transformations specified by its phases and writes the output to an intermediary file. This file is to be read by the next pass. Each method: *multi-pass* or *single-pass* has its own advantages and disadvantages. Our interpreter is a single pass interpreter.

3.2 Class Dictionary

This interpreter is implemented in C++ language. Hence C++ objects are used to represent the OCL query notation. Some classes were mentioned previously while explaining the interpretation phases. The hierarchy of nodes classes is displayed in

figure 3. This section provides a brief overview of all created classes. In the following dictionary, a class name is followed by the description of a class.

- **class Expr**
  Defined in *expr.h* file. An abstract class. Inherits from *Qualifier* class. Base class for the following classes: *Expr_string*, *Expr_num*, *Expr_subrange* and *Expr_methodcall*.

- **class Expr_methodcall**
  Defined in *expr.h* file. Inherits from *Expr* class. Used to represent method or function call expressions.

- **class Expr_num**
  Defined in *expr.h* file. Inherits from *Expr* class. Used to represent number expressions.

- **class Expr_query**
  Defined in *expr.h* file. Inherits from *Expr_subrange* class. Used to represent query expressions.

- **class Expr_special**
  Defined in *expr.h* file. Used to represent "constant" collections, e.g. `Set ''C1'', ''C2'', ''C3''` .

- **class Expr_string**
  Defined in *expr.h* file. Inherits from *Expr* class. Used to represent string expressions.

- **class Expr_subrange**
  Defined in *expr.h* file. Inherits from *Expr* class. Base class for *Expr_query* and *Expr_special* classes. Used to represent range expressions.

- **class Generator**
  Defined in *qual.h* file. Inherits from *Qualifier* class. Used to represent a generator expression, which is a particular form of a *Qualifier*.

- **class Hasclassqual**
  Defined in *qual.h* file. Inherits from *Qualifier* class.

- **class Hasclasswithqual**
  Defined in *qual.h* file. Inherits from *Qualifier* class.

- **class Localdef**
  Defined in *qual.h* file. Inherits from *Qualifier* class. Used to represent a localdef expression, which is a particular form of a *Qualifier*.

- **class OCLBag**
  Defined in *oclvalue.h* file. Inherits from *OCLCollection* class. Used to represent bag, a particular type of collection.

- **class OCLBoolean**
  Defined in *oclvalue.h* file. Inherits from *OCLValue* class. Used to represent a boolean value, e.g. true or false.

- **class OCLCollection**
  Defined in *oclvalue.h* file. Inherits from *OCLValue* class. Base class for *OCLSet*, *OCLList* and *OCLBag* classes. Used to represent collections of *OCLValue* objects.

- **class OCLList**
  Defined in *oclvalue.h* file. Inherits from *OCLCollection* class. Used to represent list, a particular type of collection.

- **class OCLNumber**
  Defined in *oclvalue.h* file. Inherits from *OCLValue* class. Used to represent numbers.

- **class OCLSet**
  Defined in *oclvalue.h* file. Inherits from *OCLCollection* class. Used to represent set, a particular type of collection.

- **class OCLString**
  Defined in *oclvalue.h* file. Inherits from *OCLValue* class. Used to represent strings.

- **class OCLValue**
  Defined in *oclvalue.h* file. An abstract class. Base class for *OCLNumber*, *OCLBoolean*, *OCLString* and *OCLCollection* classes.

- **class Qualifier**

  Defined in *qual.h* file. An abstract class. Base class for *Expr*, *Generator*, *Localdef* and *Hasclassqual* classes.

Chapter 4

Conclusion

The motivation behind this project is to study and implement OCL, Object Comprehension Language, the new powerful query notation.
The project focusses on the implementation of the interpreter of the OCL query language. The database used here is the in-memory database. The interpreter of the OCL query language is implemented in C++. It takes the advantage of the object-oriented characteristics of C++ to ease the creation of the objects and the use of their methods.
The in-memory database used for the interpreter reflects a University model, a schema of interrelated objects of classes such as Department, Staff, Student, Course, etc. The issues of error handling, user interface, and query optimization are outside the scope of this project.
Out of 28 originally suggested OCL queries, the interpreter can evaluate all but the following ones:

- Queries that require the creation of the new objects specified in a result expression (Q7).
- Nested queries (Q8).
- Queries with list elements positioning (Q22).
- Recursive queries (Q28).

These queries are left for future work. They require either recursion, and or the creation of "user-defined functions" that would also have to be interpreted.

Through the process of understanding the OCL query notation and implementing of the interpreter, valuable experience has been gained. Besides getting acquainted to the parser and scanner tools *BISON* and *FLEX*, I have to learn how to reflect properly the various types of expressions of the OCL in C++. Keeping the interpreter implementation complete, compact but flexible for any future modification as the OCL query notation evolves is also a very important principle that I have in mind. I realize that the implementation presented in this paper is not the perfect one. The C++ classes and objects have been organized into different categories according to the information they represent:

- *Parse category* reflects the expressions, queries and other information that are collected in the parsing process;
- *Value category* reflects the results of evaluating the queries and expressions and
- *Database category* reflects the database.

The *Parse category* could be done better with one class for each terminal and non-terminal in the abstract grammar.
It would also be desirable to have the OCL queries evaluated on an Ode-based database, and to have a GUI interface to allow the user to enter more than one OCL query statement at a time.

Bibliography

[1] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. Compilers, principles, techniques, and tools. Addison-Wesley, 1988.

[2] F. Bancilhon. Query languages for object-oriented database systems: Analysis and proposal. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications*, pages 1–18, Springler-Verlag, 1989.

[3] Lynn Robert Carter and William M. Waite. An introduction to compiler construction. Harper-Collins, 1993.

[4] Daniel K. C. Chan and Philip W. Trinder. Object comprehensions: A query notation for object-oriented databases. In *Proceedings of the 12th British National Conference on Databases*, Guildford, Springler-Verlag, July 1994.

[5] Servio Logic Development Corporation. Programming in opal, version 1.3. 1987.

[6] D. J. Harper, D. K. C. Chan, and P. W. Trinder. A case study of object-oriented query languages. In *Proceedings of the International Conference on Information Systems and Management of Data*, pages 63–86, Indian National Scientific Documentation Centre (INSDOC), 1993.

[7] G. Pelagatti, E. Bertino, M. Nagri, and L. Sanella. Object-oriented query languages: The notion and the issues. In *IEEE Transactions on Knowledge and Data Engineering*, pages 223–237, June 1992.

[8] P. Hudak and P. Walder. Report on the functional programming language Haskell. In *Technical Report 89/R5*, University of Glasgow, U.K., 1990.

[9] Ontologic Inc. Ontos sql guide. USA, 1991.

[10] S. Peyton-Jones. The implementation of functional programming languages. Prentice-Hall, 1987.

[11] R. Corbett, R. Stallman, and W. Hansen. The Bison reference manual version 1.25. GNU SoftWare, 1997.

[12] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. In *In Proceedings of the 1st ACM Lisp Conference*, pages 136–143, ACM Press, 1980.

[13] D. A. Turner. Recursion equation as a programming language. In *Functional Programming and its Application*, Cambridge University Press, 1981.

[14] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architectures*, pages 1–16, Springer-Verlag, 1985.

[15] V. Paxson. The Flex reference manual version 2.5. GNU SoftWare, 1995.

Appendix A

Header files: *.h

```
//expr.h
#ifndef EXPR_H
#define EXPR_H

#include <stdio.h>
#include "entity.h"
#include "DList.h"
#include "oclvalue.h"

enum Opcode {
    NoOpcode, PlusOpcode, MinusOpcode, MultOpcode, DivOpcode,
    AndOpcode, OrOpcode, DifferOpcode, UnionOpcode,
    SizeOpcode, SomeOpcode, EveryOpcode, NotOpcode,
    AtleastOpcode, JustOpcode, AtmostOpcode,
    EqOpcode, NotEqOpcode, LtOpcode, LtEqOpcode, GtOpcode, GtEqOpcode,
    DotOpcode, DotSBKOpcode, ThreedotOpcode,
    MethodcallOpcode, QueryOpcode, SpecialOpcode,
};


enum CollectionType {
    NoColType,
    SetType,
    ListType,
    BagType
};
```

```
class Expr;


class Qualifier: public anentity{
protected:
    char *qual_type;
    char *ident;
    Expr *rightexpr;
public:
    Qualifier(char*,char*id=NULL,Expr *rexpr=NULL);
    ~Qualifier();
    char *get_qualtype();
    void PrintStruct(int level = 0);
    char *get_ident();
    Expr *get_rexpr();
    void put_rexpr(Expr *e);
    virtual OCLValue *evaluate();
    virtual OCLValue *reevaluate();
    virtual char *typechecking();
};


class Expr:public Qualifier{
private:
    Opcode op;
protected:
    Expr *LeftOperand;
    Expr *RightOperand;
public:
    Expr(Expr *l=NULL,Expr *r=NULL,Opcode op1=NoOpcode);
    bool Isidentical(const Expr*);
    Expr *get_leftop() {return LeftOperand;}
    Expr *get_rightop() {return RightOperand;}
    Opcode getop() {return op;}
    void putop(Opcode op1) {op = op1;}
    virtual char *typechecking();
    virtual int IsEmpty() {}
    virtual OCLValue *evaluate();
    virtual OCLValue *dot(OCLValue*){}
    virtual int Size(){}
    virtual char *Getvarid() {}
```

```
    virtual void print_item(){}
    virtual void PrintStruct(int level = 0);
};

class Expr_string:public Expr {
protected:
    char *string_expr;
public:
    Expr_string(char *c);
    OCLValue *evaluate() {return new OCLString(string_expr);}
    char *GetType() {
      char *c = new char[MAX];
      strcpy(c, "String");
      return c;
    }
    char *typechecking() {return GetType();}
};

class Expr_num:public Expr {
protected:
    long num;
public:
    Expr_num(long i) {
       num = i;
    }
    OCLValue *evaluate() {return new OCLNumber(num);}
    char *GetType() {
      char *c = new char[MAX];
      strcpy(c, "Integer");
      return c;
    }
    char *typechecking() {return GetType();}
};


class Expr_subrange:public Expr{
//format: collection_kind LCBKTK expr THREEDOTTK expr RCBKTK
protected:
    int collect_type;
public:
    Expr_subrange(int in=NoColType,Expr *l=NULL,Expr *r=NULL,
```

```
                    Opcode op1=NoOpcode);
    int get_coltype() {return collect_type;}
    virtual char *typechecking();
    virtual OCLValue *evaluate();
};

class Expr_query:public Expr_subrange{
//format: collection_kind LSQBKTK qualifier_list BARTK expr RSQBKTK
private:
    DList<Qualifier> *qual_list;
public:
    Expr_query(DList<Qualifier> *q=NULL,int in=NoColType,Expr *l=NULL,
               Expr *r=NULL,Opcode op1=NoOpcode);
    DList<Qualifier> *Get_quallist() {return qual_list;}
    int EmptyList() {return qual_list == NULL;}
    char *typechecking();
    OCLValue *evaluate();
    anentity *gather_result(int,collection<anentity>*,char*);
};

class Expr_special:public Expr_subrange{
//format: collection_kind LCBKTK expr_list RCBKTK
private:
    DList<Expr> *expr_list;
public:
    Expr_special(DList<Expr> *e=NULL,int in=NoColType,
                 Opcode op1=NoOpcode,Expr *l=NULL,Expr *r=NULL);
    int EmptyList() {return expr_list == NULL;}
    char *typechecking();
    char *GetType() {
      char *c = new char[MAX];
      strcpy(c, "Collection");
      return c;
    }
    DList<Expr> *get_exprlist() {return expr_list;}
    OCLValue *evaluate();
};


class Expr_methodcall:public Expr{
//format: IDENTIFIERTK LBKTK expr_list RBKTK
```

```
private:
    DList<Expr> *expr_list;
    char *identifier;
public:
    Expr_methodcall(DList<Expr> *e,char *in,Opcode op1=NoOpcode,
                    Expr *l=NULL,Expr *r=NULL);
    OCLValue *evaluate(){}
};

#endif

//---------------------------- End of expr.h  -----------------------

//oclvalue.h

#ifndef OCLVALUE_H
#define OCLVALUE_H

#include <stdio.h>
#include "entity.h"
#include "DList.h"

enum CompType {
    Noreltype,
    Equaltype,
    NotEqualtype,
    LessThantype,
    LessThanEqtype,
    GreatThantype,
    GreatThanEqtype
};

enum QuantifierType {
    NoQuanType,
    SomeType,
    AtleastType,
    AtmostType,
    EveryType,
    JustType
};
```

```
class OCLNumber;
class OCL_Boolean;
class OCLString;
class OCLCollection;
class OCLSetQuant;
//class OCLValue;

class OCLValue{ //public Expr {
protected:
    char *varid;
public:
    OCLValue(char *c=NULL);
    char *Getvarid();
    void put_varid(char *c);
    virtual char *GetType();
    virtual OCLValue *evaluate();
    virtual OCLValue *plus(OCLValue*);
    virtual OCLValue *minus(OCLValue*);
    virtual OCLValue *multiply(OCLValue*);
    virtual OCLValue *divide(OCLValue*);
    virtual int Getquant();
    virtual int GetSetnum();
    virtual long Getnumb();
    virtual bool Getbool();
    virtual char* Getstr();
    virtual char* Getident();
    collection<OCLValue> *get_col() {}
    virtual int Size();
    virtual OCL_Boolean *not();
    virtual OCLValue *Compare(OCLValue*,CompType);
    virtual OCLValue *Comparenum(OCLNumber*,CompType);
    virtual OCLValue *Comparebool(OCL_Boolean*,CompType);
    virtual OCLValue *Compareset(OCLCollection*,CompType);
    virtual OCLValue *Comparequantset(OCLSetQuant*,CompType);
    virtual OCLValue *CompareString(OCLString*,CompType);
    virtual collection<anentity> *GetSetCol();
    virtual anentity *get_object();
    virtual OCLValue *dot(OCLValue*);
    virtual void print_item();
};
```

```
class OCLNumber:public OCLValue{
private:
    long numb_val;
public:
    OCLNumber(long n=0,char *c=NULL);
    char *GetType() {
char *c = new char[MAX];
strcpy(c,"OCLOCLNumber"); return c;
    }
    long Getnumb() {return this->numb_val;}
    OCLValue *evaluate();
    OCLValue *Compare(OCLValue*,CompType);
    OCLValue *Comparenum(OCLNumber*,CompType);
    OCLValue *plus(OCLValue*); /* plus ==> ADDITION*/
    OCLValue *minus(OCLValue*); /* minus ==>  SUBSTRACT*/
    OCLValue *multiply(OCLValue*); /* multiply ==>  MULTIPLICATION*/
    OCLValue *divide(OCLValue*);
    void print_item();
    void PrintStruct(int level=0);
};

class OCL_Boolean:public OCLValue{
private:
    bool bool_val;
public:
    OCL_Boolean(bool b=false,char *c=NULL);
    OCLValue *evaluate();
    char *GetType() {
        char *c = new char[MAX];
        strcpy(c,"OCLBool"); return c;
    }
    bool Getbool() {return bool_val;}
    OCL_Boolean *not();
    OCLValue *Compare(OCLValue*,CompType);
    OCLValue *Comparebool(OCL_Boolean*,CompType);
    OCLValue *plus(OCLValue*); /* plus ==> OR */
    OCLValue *multiply(OCLValue*); /* multiply ==> AND */
    void print_item();
};
```

```
class OCLString:public OCLValue{
private:
    int len;
    char *str;
public:
    OCLString();
    OCLString(char*,char *c=NULL);
    char* Getstr() {return str;}
    char *GetType() {
        char *c = new char[MAX];
        strcpy(c,"OCLString");
        return c;
    }
    OCLValue *evaluate();
    OCLValue *dot(OCLValue *e= NULL);
    OCLValue *Compare(OCLValue*,CompType);
    OCLValue *CompareString(OCLString*,CompType);
    void print_item();
    void PrintStruct(int level = 0);
};

class OCLCollection:public OCLValue{
protected:
    collection<anentity> *OCLSet_val;
    iter_list<anentity> *set_iter;
public:
    OCLCollection(collection<anentity> *C=NULL,char *c=NULL);
    collection<anentity> *GetSetCol() {return this->OCLSet_val;}
    virtual char *GetType() {
        char *c = new char[MAX];
        strcpy(c,"OCLSet"); return c;
    }
    char *Getsettype() {return set_iter->current()->GetType();}
    virtual int Size() {return (this->OCLSet_val)->size(); }
                        //call size() function of George
    OCLValue *evaluate();
    virtual OCLValue *Compare(OCLValue*,CompType);
    OCLValue *Compareset(OCLSet*,CompType);
    //OCLValue *callmethod(char* c);
    OCLValue *dot(OCLValue*); //verify if still need this function
```

```
    OCLValue *plus(OCLValue*);  /* plus ==> UNION   */
    OCLValue *minus(OCLValue*); /* minus ==> DIFFER */
    bool equal(OCLSet*);
    bool lessthan(OCLSet*);
    void print_item();
};

class OCLSet:public OCLCollection{
private:
    set<OCLValue> *setval;
public:
    OCLSet(set<OCLValue> *s = NULL) {setval = s;}
    set<OCLValue> *get_setval() {return setval;}
    collection<OCLValue> *get_col() {this->get_setval();}
}

class OCLList:public OCLCollection{
private:
    list<OCLValue> *listval;
public:
    OCLList(list<OCLValue> *l = NULL) {listval = l;}
    list<OCLValue> *get_listval() {return listval;}
    collection<OCLValue> *get_col() {this->get_listval();}
}

class OCLBag:public OCLCollection{
private:
    bag<OCLValue> *bagval;
public:
    OCLBag(bag<OCLValue> *b = NULL) {bagval = b;}
    bag<OCLValue> *get_bagval() {return bagval;}
    collection<OCLValue> *get_col() {this->get_bagval();}
}


class OCLSetQuant:public OCLCollection{
private:
    QuantifierType Qt;
    int num;
public:
    OCLSetQuant(QuantifierType Q=NoQuanType,int n=0,
```

```
collection<anentity> *C=NULL,char *c=NULL);
    OCLValue *evaluate();
    char *GetType() {
        char *c = new char[MAX];
        strcpy(c,"OCLSetQuant");
        return c;
    }
    int Getquant() {return Qt;}
    int GetSetnum() {return num;}
    OCLValue *Compare(OCLValue*,CompType);
    OCLValue *Comparequantset(OCLSetQuant*,CompType);
    void print_item(){}
};


class OCLEntity:public OCLValue{
private:
    anentity *objinstance;
public:
    OCLEntity(anentity *i = NULL,char *c = NULL);
    ~OCLEntity() {}
    char *GetType() {
        char *c = new char[MAX];
        strcpy(c,"OCLEntType"); return c;
    }
    OCLValue *evaluate();
    anentity *get_object() {return  objinstance;}
    OCLValue *dot(OCLValue*);
    void print_item() {objinstance->print_item();}
};

#endif

//---------------------------- End of oclvalue.h  ----------------------

//prog.h
#include <stdio.h>
#include <assert.h>
#include "expr.h"
#include "DList.h"
```

```
class Prog{
private:
    DList<Expr> *lstexpr;
public:
    Prog(DList<Expr> *inlist) {lstexpr = inlist;}
    int EmptyList() {return lstexpr == NULL;}
    DList<Expr> *Getlist() {return lstexpr;}
    void PrintStruct(int level = 0);
    void *evaluate();
};

//---------------------------- End of prog.h  -----------------------

//qual.h
#ifndef QUAL_H
#define QUAL_H

#include <stdio.h>
#include "expr.h"


class Generator:public Qualifier{
public:
    Generator(char *id,Expr *in_expr);
    ~Generator(){}
    char *typechecking();
    OCLValue *evaluate();
    OCLValue *reevaluate();
};

class Localdef:public Qualifier{
public:
    Localdef(char *id,Expr *in_expr);
    ~Localdef(){}
    OCLValue *evaluate();
    char *typechecking();
};

class Hasclassqual:public Qualifier{
protected:
    Expr *leftexpr;
```

```
public:
    Hasclassqual(Expr *l,Expr *r=NULL,char *id=NULL,char *typ="Hasclass");
    ~Hasclassqual(){}
    Expr *get_lexpr() {return leftexpr;}
    void put_lexpr(Expr *e) {leftexpr = e;}
    OCLValue *evaluate();
    char *typechecking();
    void Hasclassqual::calculate(collection<anentity>*,
                                 iter_list<anentity>*,char*);
};

class Hasclasswithqual:public Hasclassqual{
private:
    Expr *thirdexpr;
public:
    Hasclasswithqual(Expr *e,Expr *l,Expr *r=NULL,char *id=NULL);
    ~Hasclasswithqual(){}
    OCLValue *evaluate(){}
    char *typechecking();
};

#endif

//---------------------------- End of qual.h  -----------------------

//string.h

#ifndef STRING_H
#define STRING_H
// string.h : strings of characters
// essentially C++ book p248 et seq

#include <stream.h>
#include <string.h>

class string {
    struct srep {
        char* s;        // pointer to data
        int   lth;      // length of string
        int   rcnt;     // reference count
    };
```

```
    srep *p;

public:
    string( const char * ); //string x = "abc"
    string(); // string x;
    string( const string& ); //string x = string ...
    char* get_s() {return p->s;}
    string& operator=( const char * );
    string& operator=( const string & );
    ~string();
    char& operator[]( int i );
    const char& operator[]( int i ) const;

    friend ostream& operator<<( ostream&, const string& );
    friend istream& operator>>( istream&, string& );

    friend int operator==( const string &x, const char *s )
    { return strcmp( x.p->s, s ) == 0; }

    friend int operator==( const string &x, const string &y )
    { return (x.p->lth == y.p->lth) && (strcmp( x.p->s, y.p->s ) == 0); }

    friend int operator!=( const string &x, const char *s )
    { return strcmp( x.p->s, s ) != 0; }

    friend int operator!=( const string &x, const string &y )
    { return (x.p->lth != y.p->lth) || (strcmp( x.p->s, y.p->s ) != 0); }

    friend string concat( const string&, const string& );
    friend int hash( const string&, int size );
                                          //hash a string modulo size

};
#endif

//---------------------------- End of string.h  -----------------------
//table.h

// table.h: declaration of various types of table
// table: contains schema of the database
// hashTable: contains variables/ids read from query
```

```
// funcTable: contains functions/macros read from query
// dbtable: contains real database objects

#ifndef TABLE_H
#define TABLE_H

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "entity.h"
#include "string.h"
#include "dbase.h"

enum id_type
{
  TYPE,
  CLASS,
  METHOD,
  DATABASE
};

enum collection_type
{
  SET,
  BAG,
  LIST,
  NONE
};

class Expr;
class namenode;

class name {
    friend class table;
    friend class namenode;
public:
    name( const string& p, name* nxt );
    ~name();

    void setString( const string& s ) { str = s; }
    void setIdType( const id_type& s ) { id_kind = s; }
```

```
    void setCollectionType( const collection_type& s ) { coln = s; }
    void setBaseType( name* s ) { type_kind = s; }
    void addsuper( name* n );

    string get_str() {return str;}
    id_type get_idkind() {return id_kind;}
    collection_type get_coln() {return coln;}
    name* get_typekind() {return type_kind;}
    namenode* get_super() {return super;}
    name* get_next() {return next;}

    void print( ostream& os );
private:
    string str; //textual repn of identifier
    id_type id_kind; //grammar kind of identifier

    // type information about identifier
    collection_type coln; // kind of collection type, if any
    name* type_kind; // pointer to entry for base type
    namenode* super; // pointer to entry for super class

    name* next; //pointer to next entry in bucket
};

class namenode {
public:
    namenode( name* n, namenode* nxt ) { info = n; next = nxt; }
    ~namenode() { delete next; }
    name* get_info() {return info;}
    namenode* get_next() {return next;}

    void print( ostream& os );
private:
    name* info;
    namenode* next;
};

class table {
    name** tbl;
    int size;
```

```
public:
    table( int sz = 15 );
    ~table();

    name* look( const string& p );
    name* insert( const string& s );

    void print( ostream& os );

};

//extern table symtable;

class record {//IDENTIFIER (var read from input) structure
    char* name;
    char* type;
    //iter_list<anentity> *objiter;
    iterator<OCLValue*> it1;
    iterator<DBEntity*> it2;
    bool iter_over_db; //if iter_over_db is true then field it2 is valid

public:
    record() :name(0), type(0),it1(0), it2(0),iter_over_db(false)  {}
    record(char *n,char *t, iterator<OCLValue*> valiter = 0,
           iterator<DBEntity*> dbiter = 0, bool iterdb = true):name(n),
           type(t), it1(valiter), it2(dbiter), iter_over_db(iterdb){}
    ~record() {}
    void putName(char *n) {name = n;}
    void putNameType(char *n) {type = n;}
    void init_ocl_iterator(iterator<OCLValue*> valiter) {
      it1 = valiter;
      iter_over_db = false;
    }
    void init_dbiterator(iterator<DBEntity*> dbiter) {
      it2 = dbiter;
      iter_over_db = true;
    }
    char* getName() {return name;}
    char* getNameType() {return type;}
    OCLValue* get_curOCL() {return *it1;}
                            //return cur. obj, ptr to OCLValue
```

```
    DBEntity* get_curDB() {return *it2;}
    Whatclass??? get_curobj() { //What is the returned class ?
      if(iter_over_db) return get_curDB();
      else return get_curOCL();
    }
    void iter_next_obj() {
      if(iter_over_db) it2++; //Is it correct what I'm doing here: move
      else it1++;             //iterator to the next object in the list ?
    }
    iterator<Whatclass???> get_iterator() {//what class should I use here?
      if(iter_over_db) return it2;
      else return it1;
    }
    iter_list<anentity> *get_objiter() {return objiter;}
    void iter_nextobj() {objiter->next();} //move iterator to next object
    void reset_iter() {objiter->first();}
    anentity* get_curobj() {return objiter->current();}
};

struct storageTable {
    record* entry;
    storageTable* next;
};

class hashTable {
    storageTable **table;
    int size;

public:
    hashTable();
    ~hashTable();
    int hash(char*);
    char* look(char*);    //return type
    iter_list<anentity>* get_objiter(char*); //return iterator
    void put_objiter(char *str,iter_list<anentity> *ent);
    void insert(record*); //insert new record in hashtable
    void iter_nextobj(char*);  //move iterator of var to next object
    void reset_iter(char*); //reset iterator of var
    anentity* get_curobj(char *); //return current object of var
    void print(); // print name and type of variable
};
```

```
class funct_record {//function structure, used when read localdef;
//expression; when reading "a AS s.major.address.street", funct_record
//will have name 'a', type is basetype of 'street' method call, and
//expr_tree will point to s.major.address.street.

    char* name;
    char* type;
    Expr *expr_tree;

public:
    funct_record() :name(0), type(0),expr_tree(0) {}
    funct_record(char *n,char *t,Expr *e=NULL):name(n),
                type(t),expr_tree(e) {}
    ~funct_record() {}
    void putName(char *n) {name = n;}
    void putNameType(char *n) {type = n;}
    void putexpr(Expr *e) {expr_tree = e;}
    char* getName() {return name;}
    char* getNameType() {return type;}
    Expr *get_exprtree() {return expr_tree;}
};

struct functstorage{
    funct_record* entry;
    functstorage* next;
};

class functTable { //table store function records
    functstorage **table;
    int size;

public:
    functTable();
    ~functTable();
    int hash(char*);
    char* look(char*);    //return type
    Expr* get_exprtree(char*); //return expression tree
    void put_expr(char *str, Expr *ent);
    void insert(funct_record*); //insert new function rec in function table
    void print(); // print name and type of variable
```

```
};

class dbtable_rec {
private:
    char *tblname;//class name e.g, Course,Department,Student,Person...
    collection<anentity> *tblobj; // pointer to list of object
    dbtable_rec *next;
public:
    dbtable_rec() {tblname = NULL; tblobj = NULL;next = NULL;}
    dbtable_rec(char *c,collection<anentity> *a = NULL,
                dbtable_rec *n=NULL) {
          tblname = c; tblobj = a; next = n;
    }
    char *get_tblname() {return tblname;}
    collection<anentity> *get_set() {return tblobj;}
    dbtable_rec *get_next() {return next;}
    void put_tblname(char *c) {tblname = c;}
    void add_obj(anentity* o) {tblobj->add(o);}
    void put_next(dbtable_rec *r) {next = r;}
};

class dbtable{ // class
private:
    dbtable_rec *head;          // head pointer to list of tables
public:
    dbtable(dbtable_rec *t1=NULL) {head = t1;}
    ~dbtable() {}
    dbtable_rec *look(char*);
    void add_obj(anentity*, char*);
    void add_tbl(dbtable_rec*);
    dbtable_rec *get_head() {return head;}
    collection<anentity> *get_set(char*);
};


#endif

//---------------------------- End of table.h  -----------------------
```

Appendix B

Interpreter files: *.C

```
//expr.C
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "expr.h"
#include "oclvalue.h"
#include "table.h"

extern hashTable vartable; //contains variable read from input
                           //along with its type
extern table symtable;     //contains database schema: class names,
                           //methods,etc.

Qualifier::Qualifier(char *typ, char *id, Expr *rexpr):anentity()
{
    qual_type=new char[15];
    strcpy(qual_type,typ);
    if (id) {
        ident=new char[strlen(id)+1];
        strcpy(ident,id);
    }
    else
        ident = NULL;
        rightexpr= rexpr;
}

Qualifier::~Qualifier()
```

```
{}

char *Qualifier::get_qualtype()
{
    return qual_type;
}

char *Qualifier::get_ident()
{
    return ident;
}

Expr *Qualifier::get_rexpr()
{
    return rightexpr;
}

void Qualifier::put_rexpr(Expr *e)
{
    rightexpr = e;
}

OCLValue *Qualifier::evaluate()
{}

OCLValue *Qualifier::reevaluate()
{}

char *Qualifier::typechecking()
{}

void Qualifier::PrintStruct(int level) {
    printf("Qualifier  Type:%s, Identifier: %s\n",qual_type,ident);
    printf("Expression: ");rightexpr->print_item();printf("\n");
}

/*************************************************************************/
Expr::Expr(Expr *l, Expr *r, Opcode op1):Qualifier("Expr")
{
    LeftOperand = l;
    RightOperand = r;
```

```
    op = op1;
}

bool Expr::Isidentical(const Expr *e) // identical when both expr
                                      // have same addresses
{
    if (this == e)
        return true;
    else return false;
}


void Expr::PrintStruct(int level)
{
    if(!IsEmpty())
    {
        for(int i=0;i<level;i++)
            printf("\t");
        printf("Opcode:%d\n",op);
        for(int i=0;i<level;i++)
            printf("\t");
        printf("Left Expression:\n");
        LeftOperand->PrintStruct(level+1);
        if (RightOperand){ // unary expr does not have rightop
            for(int i=0;i<level;i++)
                 printf("\t");
            printf("Right Expression:\n");
            RightOperand->PrintStruct(level+1);
        }
    }
}

char *Expr::typechecking()
{//Recursice typechecking till the leaf level of the Expr tree, and
//ensure that type of leftop is the same as type of rightop

    Expr *left, *right;

    char *t =  new char[MAX+1];
    char *tr = new char[MAX+1];
    char *typ = new char[MAX+1];
```

```
if (LeftOperand)
    left = LeftOperand;
if (RightOperand)
    right = RightOperand;

switch(op){
  case NoOpcode: return GetType();
  case PlusOpcode: case MinusOpcode:
  case MultOpcode: case DivOpcode:
  case EqOpcode: case NotEqOpcode: case LtOpcode:
  case LtEqOpcode: case GtOpcode: case GtEqOpcode:
      strcpy(t,left->typechecking());
      strcpy(tr,right->typechecking());
      if (!strcmp(t,tr) ||
         (!strcmp(t,"String") && (!strcmp(tr,"OCLString"))) ||
         (!strcmp(tr,"String") && (!strcmp(t,"OCLString"))) ||
         (!strcmp(t,"Integer") && (!strcmp(tr,"OCLNumber"))) ||
         (!strcmp(tr,"Integer") && (!strcmp(t,"OCLNumber"))))
             return tr;
      break;

      //left should be a valid id in vartable, right should be a valid
      //method of left. Type returned is the base type of the method at
      //right if left is type String then it should exist in vartable,
      //return type recorded in vartable. If left is any base type
      //(resulted from a recursive dot expression), then verify if right
      //is a valid method of type of left.
      //Problem: for expr such as: s.address.city, s.address returns type
      //Address, thus city is a valid method of Address, it returns type
      //String. For ex.: expr such as: c.assessment.size, c.assessment
      //returns type Integer, and size is not a valid method of Integer,
      //it thus returns NULL, when in fact size is a method applies on BAG
      //of INTEGER not on INTEGER itself.
 case DotOpcode:
      t = right->typechecking();
      if (!strcmp(t,"OCLString") || !strcmp(t,"String")){
          //right should always be a string
          t = left->typechecking();
          if (!strcmp(t,"OCLString") || !strcmp(t,"String")) {
              typ = vartable.look(left->evaluate()->Getstr());
```

```
                }
                else
                    typ = t;
                if (typ){ //NON NULL type
                    string s = string(right->evaluate()->Getstr());
                    name *valid_method = symtable.look(s);
                    if ((valid_method) && (valid_method->get_idkind()==METHOD)){
                        namenode *method_class = valid_method->get_super();
                        while (method_class){
                           if(!strcmp(typ,method_class->get_info()
                                      ->get_str().get_s()))
                                //return base type
                                return valid_method->get_typekind()
                                       ->get_str().get_s();
                           else method_class = method_class->get_next();
                        }
                    }
                }
            }
            break;
        case AtleastOpcode: case AtmostOpcode: case JustOpcode:
            // leftexpr should be number, return type of rightexpr
            t = left->typechecking();
            if (!strcmp(t,"OCLNumber")|| !strcmp(t,"Integer"))
                return right->typechecking();
            break;
        case EveryOpcode: case SomeOpcode: case SizeOpcode:
            //unary expression, return type of leftexpr
            return left->typechecking();
        case NotOpcode:  //for Bool type only - unary expression
            t = left->typechecking();
            if (!strcmp(t,"OCLBool"))
                return t;
            break;
    } //end switch op

    printf("\nINVALID EXPRESSIONS\n");
    return NULL;
}

OCLValue *Expr::evaluate()
```

```
{
    int t,t2,typ;
    OCL_Boolean *b;
    OCLNumber *n;
    OCLValue *vl = LeftOperand->evaluate();
    OCLValue *vr = RightOperand->evaluate();
    OCLValue *tempval;

    switch(op){
      case NoOpcode: return NULL;
      case PlusOpcode:    // plus, union, or operations
          return (vl->plus(vr));
      case MinusOpcode:  // minus, differ operations
          return (vl->minus(vr));
      case MultOpcode:   // multiply, and operations
          return (vl->multiply(vr));
      case DivOpcode:
          return (vl->divide(vr));
      case EqOpcode:
          tempval = vl->Compare(vr,Equaltype);
          return tempval;
      case NotEqOpcode:
          return (vl->Compare(vr,NotEqualtype));
      case LtOpcode:
          return (vl->Compare(vr,LessThantype));
      case LtEqOpcode:
          return (vl->Compare(vr,LessThanEqtype));
      case GtOpcode:
          return (vl->Compare(vr,GreatThantype));
      case GtEqOpcode:
          return (vl->Compare(vr,GreatThanEqtype));
      case AtleastOpcode:
          return new OCLSetQuant(AtleastType,vl->Getnumb(),
                                 vr->GetSetCol());
      case AtmostOpcode:
          return new OCLSetQuant(AtmostType,vl->Getnumb(),
                                 vr->GetSetCol());
      case JustOpcode:
          return new OCLSetQuant(JustType,vl->Getnumb(),
                                 vr->GetSetCol());
      case EveryOpcode:
```

```
            return new OCLSetQuant(EveryType,0,vl->GetSetCol());
        case SomeOpcode:
            return (new OCLSetQuant(SomeType,0,vl->GetSetCol()));
        case DotOpcode:
            return vl->dot(vr);
        case SizeOpcode:
            return new OCLNumber(vl->Size());
        case NotOpcode:
            return (vl->not());
    }
}

Expr_string::Expr_string(char *c)
{
    int len = strlen(c);
    string_expr = new char[len+1];
    assert(string_expr != 0);
    strcpy(string_expr, c);
}

//---------------------------- End of expr.C -----------------------

//expr_der.C
#include<stdio.h>
#include "expr.h"
#include "oclvalue.h"
#include "string.h"
#include "table.h"
#include "DList.h"

extern hashTable vartable; // contains variable read from input
                           //along with its type

Expr_subrange::Expr_subrange(int in,Expr *l,Expr *r,Opcode op1)
             :Expr(l,r,op1)
{
    collect_type = in;
}


char *Expr_subrange::typechecking()
```

```
{ //format:collection_kind LCBKTK expr THREEDOTTK expr RCBKTK
  //both left & right operands should be number type and left
  //value should be smaller than right value

    Expr *left, *right;

    if (LeftOperand)
        left = LeftOperand;
    if (RightOperand)
        right = RightOperand;

    char *t;

    if (!strcmp((t = left->typechecking()),right->typechecking()))
        if (!strcmp(t,"OCLNumber")) {
            printf("\nEXPECT NUMBER IN BOTH LEFT AND RIGHT EXPRESSION\n");
            return NULL;
    }
    if (((OCLValue *)left)->Getnumb() > ((OCLValue *)right)->Getnumb()) {
        printf("\nLEFT EXPR SHOULD BE SMALLER OR EQUAL TO RIGHT EXPR\n");
        return NULL;
    }

    return t;
}

OCLValue *Expr_subrange::evaluate()
{// format: collection_kind LCBKTK expr THREEDOTTK expr RCBKTK

        collection<int> *mycol = new collection<int>();
        set<int> *myset;
        list<int> *mylist;
        bag<int> *mybag;
        OCLValue *vl = get_leftop()->evaluate();
        OCLValue *vr = get_rightop()->evaluate();

        switch(collect_type){
          case 1:
              myset = new set<int>();
              mycol = myset;
              break;
```

```
  case 2:
      mylist= new list<int>();
      mycol = mylist;
      break;
  case 3:
      mybag= new bag<int>();
      mycol = mybag;
      break;
        }
        return new OCLEntity(mycol);
}

/*************************************************************************/

Expr_query::Expr_query(DList<Qualifier> *q,int in,Expr *l,
                       Expr *r,Opcode op1):Expr_subrange(in,l,r,op1)
{
    qual_list = q;
}

char *Expr_query::typechecking()
{//format: collection_kind LSQBKTK qualifier_list BARTK expr RSQBKTK

    if (EmptyList()) {
        printf("\nINVALID EMPTY EXPRESSION\n");
        return NULL;
    }

    // The 1st qualifier in the qualifier_list must be of type generator
    qual_list->Reset(); // Reset to the beginning of the list
    Qualifier *current = qual_list->GetCur();
    if (strcmp(current->get_qualtype(),"Generator")){
        // if not type Generator print error message
        printf("\nEXPECT GENERATOR AS FIRST QUALIFIER IN QUALIFIER LIST\n");
        return NULL;
    }

    //typecheck each qualifier of qualifier_list
    while(current=qual_list->GetCur()){
        if(!current->typechecking()) { //NULL type
            printf("\nINVALID EXPRESSION\n");
```

```
            return NULL;
        }
        current = qual_list->GoNext();
    }
    //typecheck the returned expression
    char *t = get_leftop()->typechecking();
    if(!t) {
        printf("\nINVALID RETURNED EXPRESSION\n");
        return NULL;
    }

    return t;
}

OCLValue *Expr_query::evaluate()
{//format: collection_kind LSQBKTK qualifier_list BARTK expr RSQBKTK

//evaluate each member of qual_list; if it returns true, continue
//evaluating next member; if it returns false, for Generator: backup
//to the previous Generator to evaluate the previous Generator again
//(fetch next object); for filter: backup to the previous Generator
//that matches the id, evaluate the Generator again, then evaluate the
//filter; continue this process until filter returns true; when finish
//one round, if true for all filters, add object to the set;

//A Generator returns an OCLEntity which contains object/dbinstance
//A filter returns OCLBoolean which contains boolean value and a varid

    Qualifier *current, *tmpqual;
    collection<anentity> *result = new collection<anentity>();
    OCLValue *gen_val, *filter_val;
    int done, finished = 0;
    char *rettype;

    qual_list->Reset(); // Reset to the beginning of the qual_list
    while (!finished) {
        done = 0;
        while(current=qual_list->GetCur()) {
            if(!strcmp(current->get_qualtype(),"Generator")) {
                //is a Generator
                gen_val = current->evaluate();
```

```
    if(!gen_val->get_object()) {//Generator returns NULL obj
        done = 0;
        current->reevaluate(); //reset the iterator of the
                               //current Gen.
        tmpqual = qual_list->GoPrev();  //backup to the previous
                                        //Generator
        while(tmpqual && strcmp(tmpqual->get_qualtype(),
               "Generator")) //not a Gen.
             tmpqual = qual_list->Prev();
        if (!tmpqual){ //1st generator of the list returns NULL
            finished = 1; //get out of the loop
            break;
        }
    }
    else { //Generator returns valid obj
        done = 1;
        current = qual_list->GoNext();
    }
}
else { //is a filter
    filter_val = current->evaluate();
    if (!filter_val->Getbool()) { //filter returns false
        done = 0;
        tmpqual = qual_list->GoPrev(); //backup
        if (!filter_val->Getvarid())   //NULL id: backup to the
                                       //prev Gen
            while(tmpqual && strcmp(tmpqual->get_qualtype(),
                  "Generator"))
                tmpqual = qual_list->GoPrev();
        else //No NULL id: backup to the prev Gen that matches id
            while(tmpqual &&
                  strcmp(tmpqual->get_qualtype(),"Generator") &&
                  strcmp(tmpqual->get_ident(),
                         filter_val->Getvarid()))
                  tmpqual = qual_list->GoPrev();
        if (tmpqual->evaluate()->get_object())//evaluate generator
                                              //node
            while (qual_list->GoNext() != current)
                   ;
        else {
            done = 0;
```

```
                        break;
                    }
                }
                else { //filter returns true
                    done = 1;
                    current = qual_list->GoNext();
                }
            }
        } //end current
        if(done) {
            //evaluate the returned expression
            OCLValue *val_ret = get_leftop()->evaluate();
            if(val_ret->Getvarid())//returned expr is not a simple character
                rettype = val_ret->GetType();
            else {
                result->add(gen_val->get_object());
                rettype = vartable.look(val_ret->Getstr());
            }
            qual_list->Reset(); //start over from the 1st generator to
                                //fetch new object
        }
    } //end finished

    return new OCLEntity(gather_result(get_coltype(),result,rettype));
}


anentity *Expr_query::gather_result(int coltype,
                                    collection<anentity> *mycol,
                                    char *returntype)
{
    set<person> *mysp = new set<person>();
    set<course> *mysc = new set<course>();
    list<person> *mylp = new list<person>();
    list<course> *mylc = new list<course>();
    bag<person> *mybp = new bag<person>();
    bag<course> *mybc = new bag<course>();
    iter_list<anentity> *myiter = new iter_list<anentity>(mycol);

    if(!strcmp(returntype,"Person"))
       switch(coltype){
```

```
      case 1: //Set
         //move instance from mycol list to myfinal list
          while (myiter->POS() != NULL){ /* current is not null */
              mysp->add((person*) (void *)myiter->current());
              myiter->next();
          }
          return mysp; //collection cannot be wrapped in an OCLSet
      case 2: //List
          //move instance from mycol list to myfinal list
          myiter->first(); /* reset to the beginning of the list */
          while (myiter->POS() != NULL){ /* current is not null */
              mylp->add((person*) (void *)myiter->current());
              myiter->next();
          }
          return mylp;
      case 3: //Bag
          //move instance from mycol list to myfinal list
          myiter->first(); /* reset to the beginning of the list */
          while (myiter->POS() != NULL){ /* current is not null */
              mybp->add((person*) (void *)myiter->current());
              myiter->next();
          }
          return mybp;
   } //end switch
else if(!strcmp(returntype,"Course"))
   switch(coltype){
     case 1: //Set
        //move instance from mycol list to myfinal list
         myiter->first(); /* reset to the beginning of the list */
         while (myiter->POS() != NULL){ /* current is not null */
             mysc->add((course*) (void *)myiter->current());
             myiter->next();
         }
         return mysc;
      case 2: //List
          //move instance from mycol list to myfinal list
          myiter->first(); /* reset to the beginning of the list */
          while (myiter->POS() != NULL){ /* current is not null */
              mylc->add((course*) (void *)myiter->current());
              myiter->next();
          }
```

```
            return mylc;
        case 3: //Bag
            //move instance from mycol list to myfinal list
            myiter->first(); /* reset to the beginning of the list */
            while (myiter->POS() != NULL){ /* current is not null */
                mybc->add((course*) (void *)myiter->current());
                myiter->next();
            }
            return mybc;
        }
}

/*************************************************************************/

Expr_special::Expr_special(DList<Expr> *e,int in,Opcode op1,Expr *l,
                           Expr *r):Expr_subrange(in,l,r,op1)
{
    expr_list = e;
}


char *Expr_special::typechecking()
{//format: collection_kind LCBKTK expr_list RCBKTK
 //ensure each expression in the expr_list has the same type
 //and return its type.

  char *type1, *type2;
  Expr *current;
  expr_list->Reset();
  current = expr_list->GetCur();
  type1 = current->typechecking();
  current = expr_list->GoNext();
  while(current = expr_list->GetCur()) {
    if ((type2 = current->typechecking()) && !strcmp(type1,type2))
      current = expr_list->GoNext();
    else {
      printf("Expressions are not of the same type\n");
      return NULL;
    }
  }
  return type1;
```

```
}


OCLValue *Expr_special::evaluate()
{//format: collection_kind LCBKTK expr_list RCBKTK
 //return OCLSet

  Expr *current;
  set<OCLValue> *myset = new set<OCLValue>();
  list<OCLValue> *mylist = new list<OCLValue>();
  bag<OCLValue> *mybag = new bag<OCLValue>();

  switch(get_coltype()){
    case 1: //Set
      expr_list->Reset();
      while(current = expr_list->GetCur()) {
          myset->add(current->evaluate());
          current = expr_list->GoNext();
      }
      return new OCLSet(myset);
      //should have OCLSet with set<OCLValue> as member
    case 2: //List
      expr_list->Reset();
      while(current = expr_list->GetCur()) {
          mylist->add(current->evaluate());
          current = expr_list->GoNext();
      }
      return new OCLList(mylist); //should have OCLList
    case 3: //Bag
      expr_list->Reset();
      while(current = expr_list->GetCur())
          mybag->add(current->evaluate());
          current = expr_list->GoNext();
      }
      return new OCLBag(mybag); //should have OCLBag
      //end switch
}

/*************************************************************************/

Expr_methodcall::Expr_methodcall(DList<Expr> *e,char *in,
```

```
 Opcode op1,Expr *l,Expr *r):Expr(l,r,op1)
{
    expr_list = e;
    identifier=new char[strlen(in)+1];
    strcpy(identifier,in);
}

//---------------------------- End of expr_der.C ----------------------

//main.C
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "string.h"
#include "token.h"
#include "error.h"
#include "table.h"

table symtable(50); //schema of database
hashTable vartable; //variables/ids read from query
functTable macrotable; //macro read from query
dbtable database;

#include "lex.h"
#include "syn.h"
#include "prog.h"
#include "expr.h"
#include "entity.h"


extern FILE *yyin;
extern Prog *program;

int yyparse(void);

int main(int argc, char **argv)
{
   yyin = NULL;

   if (argc != 3){
       printf("\nUsage: thetest <schema filename> <query filename> \n");
```

```
        exit(1);
    }


    //read schema first
    ifstream f_in(argv[1]);
    if (!f_in){
        printf("\nFile %s cannot be opened\n", argv[1]);
        exit(1);
    }

    //read query last
    if ((yyin=fopen(argv[2], "r"))==NULL) {
        printf("\nFile %s can not be opened\n", argv[2]);
        exit(1);
    }

    //parse schema
    curr_tok = get_token(f_in);
    schema_def(f_in);
    //return no_of_errors;

    //create dbtable database with sample objects
    person *p1 = new person("Bob","111 Yoyo","Florida");
    person *p2 = new person("John","123 Palm","Texas");
    collection<anentity> *setperson = new collection<anentity>();
    setperson->add(p1);
    setperson->add(p2);
    dbtable_rec *d= new dbtable_rec("Person",setperson); //database objects
    database.add_tbl(d);

    //parse query
    yyparse();
    fclose(yyin);
    program->evaluate();
    program->PrintStruct();
}

//---------------------------- End of main.C -----------------------

//prog.C
```

```
#include <stdio.h>
#include "prog.h"
#include "DList.h"
#include "expr.h"

void Prog::PrintStruct(int level)
{
    for(int i=0;i<level;i++)
        printf("\t");
    printf("PROGRAM:");
    if (!EmptyList())
    {
        Expr *anexpr;
        lstexpr->Reset();
        int j=1;
        while(anexpr=lstexpr->Next())
        {
            printf("\n");
            for(int i=0;i<level+1;i++)
               printf("\t");
            printf("Expression #%d:\n",j++);
            anexpr->PrintStruct(level+2);
        }
    }
    printf("\n");
}

void *Prog::evaluate()
{
    Expr *curexpr;
    if (!EmptyList()){
        this->lstexpr->Reset();
        while(curexpr=lstexpr->Next()){
            char *type = curexpr->typechecking();
            if (type) {
               OCLValue *v = curexpr->evaluate();
               v->print_item();
            }
        }
    }
```

```
}

//---------------------------- End of prog.C ----------------------

//qual.C
#include<stdio.h>
#include<stl.h>  //for built-in iterator
#include "qual.h"
#include "expr.h"
#include "oclvalue.h"
#include "table.h"

extern hashTable vartable;// contains variable read from input along
                          //with its type
extern table symtable;   // contains database schema: class names, methods ...
extern dbtable database; // contains the real database
extern functTable macrotable; //contains macro name and expression tree

/*************************************************************************/
Generator::Generator(char *id,Expr *in_expr)
          :Qualifier("Generator",id,in_expr) {}

char *Generator::typechecking()
{//format: IDENTIFIER LIMPLTK expression (LIMPLTK: <- )

    //IDENTIFIER should not exist in vartable.
    //if rightexpr is of type STRING EXPRESSION, then it should
    //exist in dbtable database; return type of right expression
    //In addition to return the type, typechecking also set up the
    //IDENTIFIER and its type in vartable.

    name *dbname;
    char *id = get_ident();
    if (vartable.look(id)) {
        printf("\nIDENTIFIER %s ALREADY DEFINED EARLIER\n",id);
        return NULL;
    }

    //typecheck right expression
    char *typ = get_rexpr()->typechecking();
    if (!typ){
```

```
        printf("\nINVALID EXPRESSION %s\n",typ);
        return NULL;
    }
    //right expression can be a single String (e.g Departments,
    //Students,etc.), or a collection of String, or type resulted
    //from a dot expression (e.g. c <- s.takes, right expression
    //returns type Course). If it's a single string, then verify
    //if it exists in symtable, database schema.

    if (!strcmp(typ,"String")) {
        if(!strcmp("String",get_rexpr()->GetType())) {//single String
            string s = string(get_rexpr()->evaluate()->Getstr());
            if((dbname = symtable.look(s)) &&
               (dbname->get_idkind()==DATABASE))
                typ = dbname->get_typekind()->get_str().get_s();
            else {
               printf("\nINVALID EXPRESSION %s\n",
               get_rexpr()->evaluate()->Getstr());
               return NULL;
            }
        }
    }

    record *newrec = new record(id,typ);
    vartable.insert(newrec);
    return typ;
}


OCLValue *Generator::evaluate()
{//format: IDENTIFIERTK LIMPLTK expr

    //IDENTIFIER exists in vartable, if its iterator is NULL,
    //get its type and setup its iterator. If its iterator is
    //not NULL, move iterator to next object. Return current
    //object of iterator after setup or after moved.

    char *id = get_ident();
    string s = string(vartable.look(id)); //return type of id then
                                          //convert it to string
    if(!symtable.look(s)){ //if type of id does not exist in symtable
```

```
        if (!vartable.get_objiter(id)) { //iterator is NULL
            OCLValue *v = get_rexpr()->evaluate();
            OCLValue::iterator it = v->get_col()->begin();
            vartable.put_objiter(id,it);
            return new OCLEntity(*it); //*it returns current object of it
        }
        else {//iterator is not NULL, move it to next object
            it++; //??not sure about this
            return new OCLEntity(*it);
        }
    else { //type of id exist in symtable
       if (!vartable.get_objiter(id)) { //iterator is NULL
            iter_list<anentity> *setiter= new iter_list<anentity>
               (database.get_set(vartable.look(id))); //create iterator
            vartable.put_objiter(id,setiter); //set up iterator for id
                                              //in vartable
            return new OCLEntity(setiter->current());
        }
        else { //iterator is not NULL
            vartable.iter_nextobj(id); //move it to next obj
            return new OCLEntity(vartable.get_curobj(id));
        }
    }
}

OCLValue *Generator::reevaluate()
{//reset the iterator of the variable in the vartable to the beginning

    char *id = get_ident();
    if (vartable.get_objiter(id)) { //for caution only: iterator is not NULL
        vartable.reset_iter(id);
        return new OCLEntity(vartable.get_curobj(id));
    }
    return NULL;
}

/*************************************************************************/
Localdef::Localdef(char *id,Expr *in_expr):Qualifier("Localdef",id,in_expr)
{}

char *Localdef::typechecking()
```

```
{//format: IDENTIFIERTK ASTK expr
//IDENTIFIER should not exist in vartable, insert IDENTIFIER in macrotable
//along with the type returned from expr->typechecking(); and a pointer to
//expression tree

    char *id = get_ident();
    if (vartable.look(id)) {
        printf("\nIDENTIFIER %s ALREADY DEFINED EARLIER\n",id);
        return NULL;
    }

    //typecheck right expression
    char *typ = get_rexpr()->typechecking();
    if (!typ){
        printf("\nINVALID EXPRESSION %s\n",typ);
        return NULL;
    }

    funct_record *newrec = new funct_record(id,typ,get_rexpr());
    macrotable.insert(newrec);
    return typ;
}

OCLValue *Localdef::evaluate()
{//format: IDENTIFIERTK ASTK expr
//Don't know what to do here, since 'expr' is evaluated only when
//IDENTIFIER is encountered in a normal expression

    return new OCL_Boolean(true);

}


/*************************************************************************/
Hasclassqual::Hasclassqual(Expr *l,Expr *r,char *id,
                           char *typ="Hasclass"):Qualifier(typ,id,r)
{
    leftexpr = l;
}

char *Hasclassqual::typechecking()
```

```
{ // Hasclassqual format: expr HASCLASSTK expr;
// leftexpr should exist in vartable with returned type T
// Verify if type requested in query (rightexpr) is indeed a derived
// class of T; look for rightexpr in symtable, if found, compare it with
// type T returned from searching for leftexpr in vartable, if T is indeed
// a superclass of rightexpr,then modify leftexpr and rightexpr pointers
// to the evaluated values (vl and vr below)

    OCLValue *vl = get_lexpr()->evaluate();
    OCLValue *vr = get_rexpr()->evaluate();
    namenode *c;
    name* n;
    char *id, *typ1, *typ2;
    int done = 0;
    if(!strcmp(vl->GetType(),"OCLString") &&
       !strcmp(vr->GetType(),"OCLString")){
        id = vl->Getstr();
        typ2 = vr->Getstr();      // type requested from queries
        if ((typ1=vartable.look(id)) && (n = symtable.look(string(typ2))))
            while((c=n->get_super()) && !done) //n has superclass
                if (!strcmp(typ1,c->get_info()->get_str().get_s()))
                        // if left is superclass of right
                    return typ2;
                else n = c->get_info(); //move n one level up in the ancestor
                                        //hierarchy
        // type of leftexpr is not a supertype of rightexpr
        printf("\n INVALID EXPRESSIONS: %s \n",typ2);
        return NULL;
    }
    printf("\n HASCLASS ACCEPTS STRING EXPRESSIONS ONLY\n");
    return NULL;
}

OCLValue *Hasclassqual::evaluate()
{  // Hasclassqual format: expr HASCLASSTK expr

    // locate type of leftexpr in database to get to the set of database
    // traverse the set and collect member that is the object of rightexpr

    char *id = get_lexpr()->evaluate()->Getstr();
    char *typ1 = vartable.look(id);        // type recorded in vartable for
```

```
                                                    //leftexpr
    char *typ2 = get_rexpr()->evaluate()->Getstr();  // type requested from
                                                     //queries
    iter_list<anentity> *iterat=new iter_list<anentity>
                                                (database.get_set(typ1));
    collection<anentity> *mycollection = new collection<anentity>();
    calculate(mycollection,iterat,typ2);
    if (!(mycollection->is_empty()))
        return new OCLSet(mycollection);
    return 0;
}

void Hasclassqual::calculate(collection<anentity> *mycol,
                  iter_list<anentity> *iter,char *c)
{
// traverse each member of the iterator 'iter', if it has type c
// then add it to mycol

    char *t;
    iter->first(); // reset to the beginning of the list
    while (iter->POS() != NULL){ //invoke 'GetType' method on each member
        t = iter->current()->GetType();
        if (!strcmp(c,t))
           mycol->add(iter->current());
        iter->next();
    }
}

/*************************************************************************/
Hasclasswithqual::Hasclasswithqual(Expr *e,Expr *l,Expr *r,
                 char *id):Hasclassqual(l,r,id,"Hasclasswith")
{
    thirdexpr = e;
}

char *Hasclasswithqual::typechecking()
{// format: expr HASCLASSTK expr WITHTK expr

    return (Hasclassqual::typechecking());
}
```

```
//---------------------------- End of qual.C -----------------------

//string.C
// string.c : strings of characters
// essentially C++ book p184 et seq

#include <stream.h>
#include "error.h"
#include "string.h"

string::string( const char* s )
{
        p = new srep;
        p->lth = strlen(s);
        p->s = new char[p->lth+1];
        strcpy( p->s, s );
        p->rcnt = 1;
}

string::string()
{
        p = new srep;
        p->lth = 0;
        p->s = 0;
        p->rcnt = 1;
}

string::string( const string& x )
{
        x.p->rcnt++;
        p = x.p;
}

string& string::operator=( const char* s )
{
        if ( p->rcnt > 1 ) {//disconnect self
             p->rcnt--;
             p = new srep;
        }
        else if ( p->rcnt == 1)
             delete[] p->s;
```

```
        p->lth = strlen(s);
        p->s = new char[p->lth+1];
        strcpy( p->s, s );
        p->rcnt = 1;
        return *this;
}

string& string::operator=( const string& x )
{
        x.p->rcnt++;//protect against ``st = st''
        if ( --p->rcnt == 0 ) {
            delete[] p->s;
            delete p;
        }
        p = x.p;
        return *this;
}

string::~string()
{
        if( p->rcnt == 0 )
            error( "destructing string with zero reference count" );
        // cout << "<destructing string: " << *this << ">" << endl;
        if ( --p->rcnt == 0 ) {
            // cout << "   really" << endl;
            delete[] p->s;
            delete p;
        }
}

char& string::operator[]( int i )
{
        if( i < 0 || p->lth < i ) error( "index out of range" );
        return p->s[i];
}

const char& string::operator[]( int i ) const
{
        if( i < 0 || p->lth < i ) error( "index out of range" );
        return p->s[i];
```

```
}

ostream& operator<<( ostream& s, const string& x )
{
        return( s << x.p->s );
}

istream& operator>>( istream& s, string& x )
{
        char buf[256];
        s >> buf;
        x = buf;//using string assignment
        return s;
}

string concat( const string& s1, const string& s2 )
{
        int lth = s1.p->lth + s2.p->lth;
        char* s = new char[lth+1];
        strcpy( s, s1.p->s );
        strcpy( s + s1.p->lth, s2.p->s );
        return string( s );
}

int hash( const string& x, int size )
{
        int ii = 0;
        char* pp = x.p->s;

        while( *pp ) ii = ii << 1 ^ *pp++;
        if ( ii < 0 ) ii = -ii;
        return ii % size;
}

//---------------------------- End of string.C -----------------------

//table.C
// table.c : symbol table declarations
// C++ book page 155 and 156
//   modified to use string class
```

```
#include <stdlib.h>
#include <string.h>
#include "error.h"
#include "string.h"
#include "table.h"
#include "expr.h"

#define HASHSIZE 21

table::table( int sz )
{
        if( sz < 0 ) error( "negative table size" );
        tbl = new name*[ size = sz ];
        for ( int i = 0; i < sz; i++ ) tbl[i] = 0;
}

table::~table()
{
        for ( int i = 0; i < size; i++ ){
           name* nx;
           for ( name* n = tbl[i]; n;  n = nx) {
                 nx = n -> next;
                 delete n;
           }
         }
         delete tbl;
}

name* table::look( const string& p )
{
         int ii = hash( p, size );

         for( name* n = tbl[ii]; n; n = n->next )//search
              if ( p == n->str ) return n;

         //not found, so return null pointer
         return 0;
}

name* table::insert( const string& p )
{
```

```
        int ii = hash( p, size );

        for( name* n = tbl[ii]; n; n = n->next )//search
             if ( p == n->str ) return n;

        name* nn = new name( p, tbl[ii] );
        tbl[ii] = nn;
        return nn;
}

void namenode::print( ostream& os )
{
        os << info->str;
        if( next )
        {
          os << ",";
          next->print( os );
        }
}

name::name( const string& p, name* nxt )
{
        str = p;
        id_kind = CLASS;
        coln = NONE;
        type_kind = 0;
        super = 0;
        next = nxt;
}

name::~name( )
{
        delete super;
}

void name::addsuper( name* n )
{
        super = new namenode( n, super );
}
void name::print( ostream& os )
{
```

```
        os << str << ":\t\t";
        switch( id_kind ){
           case TYPE: os << "TYPE "; break;
           case CLASS: os << "CLASS "; break;
           case METHOD: os << "METHOD "; break;
           case DATABASE: os << "DATABASE ";
        }
        switch( coln){
           case SET: os << "SET "; break;
           case BAG: os << "BAG "; break;
           case LIST: os << "LIST "; break;
           case NONE: os << " ";
        }
        if ( type_kind )
        {
           os << " BaseType(" << type_kind->str << ")";
        }
        if ( super )
        {
           os << " SuperType("; super->print( os ); os << ")";
        }
        os << endl;
}

void table::print( ostream& os )
{
        for ( int i = 0; i < size; i++ ){
           name* nx;
           for ( name* n = tbl[i]; n;  n = nx) {
                 nx = n -> next;
                 n->print( os );
           }
        }
}
//*****************************************************************

hashTable::hashTable()
{
    table = new storageTable* [size = HASHSIZE];
    for (int i = 0; i < size; i++)
         table[i] = 0;
```

```
}

hashTable::~hashTable()
{
    for (int i = 0; i < size; i++)
    {
        storageTable* t;
        storageTable* tt;
        for (t = table[i]; t; t = tt)
        {
            tt = t->next;
            delete t;
        }
    }
    delete table;
}

int hashTable::hash(char *str)
{
    unsigned i;// to ensure that the hash value is non-negative

    for (i = 0; *str; str++)
         i = *str + 22 * i;

    return i % size;
}

char* hashTable::look(char *str)
{
    for (storageTable* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
           return t->entry->getNameType();
    return 0;
}

iter_list<anentity>* hashTable::get_objiter(char *str)
{
    for (storageTable* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
            return t->entry->get_objiter();
    return 0;
```

```
}

void hashTable::put_objiter(char *str,iter_list<anentity> *ent)
{
    for (storageTable* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
            t->entry->putObject(ent);
}

void hashTable::iter_nextobj(char *str)
{
    for (storageTable* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
            t->entry->iter_nextobj();
}

void hashTable::reset_iter(char *str)
{
    for (storageTable* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
            t->entry->reset_iter();
}

anentity* hashTable::get_curobj(char *c)
{
    for (storageTable* t = table[hash(c)]; t; t = t->next)
        if (!strcmp(c, t->entry->getName()))
           return t->entry->get_curobj();
    return 0;
}

void hashTable::insert(record *rec)
{
    if(look(rec->getName())) return; //don't insert - it's already in
    int i;       //insert otherwise
    storageTable* t = new storageTable;
    t->entry = rec;
    t->next = table[i = hash(rec->getName())];
    table[i] = t;
}
```

```
void hashTable::print()
{
    int i;
    for (i = 0; i <= size; i++)
        for (storageTable* t = table[i]; t; t = t->next)
            cout << "Id ( "<<t->entry->getName()<< " )
                of Type: "<< t->entry->getNameType() <<endl;
}
//*************************************************************************

functTable::functTable()
{
    table = new functstorage* [size = HASHSIZE];
    for (int i = 0; i < size; i++)
        table[i] = 0;
}

functTable::~functTable()
{
    for (int i = 0; i < size; i++)
    {
        functstorage* t;
        functstorage* tt;
        for (t = table[i]; t; t = tt)
        {
            tt = t->next;
            delete t;
        }
    }
    delete table;
}

int functTable::hash(char *str)
{
    unsigned i;  // to ensure that the hash value is non-negative
    for (i = 0; *str; str++)
        i = *str + 22 * i;
    return i % size;
}
```

```
char* functTable::look(char *str)
{
    for (functstorage* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
           return t->entry->getNameType();
    return 0;
}

Expr* functTable::get_exprtree(char *str)
{
    for (functstorage* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
           return t->entry->get_exprtree();
    return 0;
}


void functTable::put_expr(char *str,Expr *e)
{
    for (functstorage* t = table[hash(str)]; t; t = t->next)
        if (!strcmp(str, t->entry->getName()))
            t->entry->putexpr(e);
}

void functTable::insert(funct_record *rec)
{
    if(look(rec->getName())) return; //don't insert - it's already in
    int i;  //insert otherwise
    functstorage* t = new functstorage;
    t->entry = rec;
    t->next = table[i = hash(rec->getName())];
    table[i] = t;
}


void functTable::print()
{
    int i;

    for (i = 0; i <= size; i++)
        for (functstorage* t = table[i]; t; t = t->next) {
```

```
            cout << "Id ( " <<t->entry->getName()<< " )
              of Type: " <<t->entry->getNameType()<<endl;
        }
}

//*************************************************************************

dbtable_rec *dbtable::look(char *c)
{
    for(dbtable_rec *db = head; db; db = db->get_next()){
        char *t = db->get_tblname();
        if(!strcmp(t,c)) return db;
    }
    return 0;
}

void dbtable::add_obj(anentity *o, char *c)
//add object 'o' to the table named 'c'
{
    if (dbtable_rec *d = look(c))
        d->add_obj(o);
    else
        printf("INVALID TABLE NAME\n");
}

void dbtable::add_tbl(dbtable_rec *t) //add new table
{
    int found = 0;
    dbtable_rec *tmp;
    if (!head) //table is empty
        head = t;
    else {
        for (tmp = head; tmp; tmp = tmp->get_next()) {
            char *t1 = tmp->get_tblname();
            char *t2 = t->get_tblname();
            if(!strcmp(t1,t2)){ //search, if found do not add
                found = 1;
                break;
            }
        }
        if (!found){ // if not found, add to the end of the list
```

```
            tmp->put_next(t);
            t->put_next(NULL);
        }
    }
}

collection<anentity> *dbtable::get_set(char *c)
//get list of object of table 'c'
{
    if (dbtable_rec *d = look(c))
        return d->get_set();
    return NULL;
}

//---------------------------- End of table.C -----------------------

//value.C
#include<stdio.h>
#include<assert.h>
#include "expr.h"
#include "oclvalue.h"
#include "table.h"

extern hashTable vartable; //contains variable read from input
                           //along with its type
extern functTable macrotable; //contains macro name and expression tree

OCLValue::OCLValue(char *c=NULL)
{
    varid=c;
}

char *OCLValue::Getvarid()
{
    return varid;
}

void OCLValue::put_varid(char *c)
{
    varid = c;
}
```

```
char *OCLValue::GetType() {}

OCLValue *OCLValue::evaluate() {}

OCLValue *OCLValue::plus(OCLValue *v) {}

OCLValue *OCLValue::minus(OCLValue *v) {}

OCLValue *OCLValue::multiply(OCLValue *v) {}

OCLValue *OCLValue::divide(OCLValue *v) {}

int OCLValue::Getquant() {}

int OCLValue::GetSetnum() {}

long OCLValue::Getnumb() {}

bool OCLValue::Getbool() {}

char* OCLValue::Getstr() {}

char* OCLValue::Getident() {}


int OCLValue::Size() {}

OCL_Boolean* OCLValue::not() {}

OCLValue *OCLValue::Compare(OCLValue *e, CompType t) {}

OCLValue *OCLValue::Comparenum(OCLNumber *n, CompType t) {}

OCLValue *OCLValue::Comparebool(OCL_Boolean *b, CompType t){}

OCLValue *OCLValue::Compareset(OCLCollection * , CompType t) {}

OCLValue *OCLValue::Comparequantset(OCLSetQuant *s, CompType t) {}

OCLValue *OCLValue::CompareString(OCLString *s, CompType t) {}
```

```
collection<anentity> *OCLValue::GetSetCol() {}

anentity *OCLValue::get_object() {}

OCLValue *OCLValue::dot(OCLValue *v) {}

void OCLValue::print_item() {}

/*************************************************************************/
OCLNumber::OCLNumber(long n,char *c):OCLValue(c)
{
    numb_val = n;
}

OCLValue *OCLNumber::evaluate()
{
    OCLNumber *n = new OCLNumber(this->numb_val);
    return n;
}

OCLValue *OCLNumber::Compare(OCLValue *e,CompType C)
{// e: Rightoperand ; this: Leftoperand
    return(e->Comparenum(this,C));
}

OCLValue *OCLNumber::Comparenum(OCLNumber *num,CompType C)
{// num: Leftoperand ; this: Rightoperand
    bool result = 0;
    switch(C){
      case Equaltype:
          if (this->numb_val == num->numb_val) result = 1;
          break;
      case GreatThantype:
          if (num->numb_val > this->numb_val) result = 1;
          break;
      case LessThantype:
          if (num->numb_val < this->numb_val) result = 1;
          break;
      case LessThanEqtype:
          if ((num->numb_val < this->numb_val) ||
```

```
            (this->numb_val == num->numb_val))
            result = 1;
        break;
    case GreatThanEqtype:
        if ((num->numb_val > this->numb_val) ||
        (this->numb_val == num->numb_val))
            result = 1;
        break;
    }
    if (!this->Getvarid())
        return new OCL_Boolean(result,num->Getvarid());
    else
        return new OCL_Boolean(result);
}


OCLValue *OCLNumber::plus(OCLValue *num)
{
    this->numb_val += num->Getnumb();
    return this;
}


OCLValue *OCLNumber::minus(OCLValue *num)
{
    this->numb_val -= num->Getnumb();
    return this;
}

OCLValue *OCLNumber::multiply(OCLValue *num)
{
    this->numb_val *= num->Getnumb();
    return this;
}

OCLValue *OCLNumber::divide(OCLValue *num)
{
    if (num->Getnumb() != 0){
        this->numb_val /= num->Getnumb();
        return this;
    }
```

```
}

void OCLNumber::print_item()
{
    cout << "OCLNumber: " << numb_val << "\n";
}

void OCLNumber::PrintStruct(int level = 0)
{
    //Expr::PrintStruct(level);
    for(int i=0;i<level;i++)
        printf("\t");
    printf("OCLNumber Expression:");
    print_item();
}
/*************************************************************************/

OCL_Boolean::OCL_Boolean(bool b,char *c):OCLValue(c) /
{
    bool_val = b;
}


OCLValue *OCL_Boolean::evaluate()
{
    return new OCL_Boolean(this->bool_val);
}

OCL_Boolean *OCL_Boolean::not()
{
    this->bool_val = !(this->bool_val);
    return this;
}

OCLValue *OCL_Boolean::Compare(OCLValue *e,CompType C)
{//e: Rightoperand ; this: Leftoperand
    if (C == Equaltype)
        return(e->Comparebool(this,C));
    return new OCL_Boolean(false);
}
```

```
OCLValue *OCL_Boolean::Comparebool(OCL_Boolean *b,CompType C)
{//b: Leftoperand
    bool result = false;
    if (this->bool_val == b->bool_val)
       result = true;
    if (!this->Getvarid())
       return new OCL_Boolean(result,b->Getvarid());
    else
       return new OCL_Boolean(result);
}

OCLValue *OCL_Boolean::plus(OCLValue *b) /* OR operation */
{
    this->bool_val = this->bool_val || b->Getbool();
    return this;
}


OCLValue *OCL_Boolean::multiply(OCLValue *b) /* AND operation */
{
    this->bool_val = this->bool_val && b->Getbool();
    return this;
}

void OCL_Boolean::print_item()
{
    if (bool_val)
       cout << "Boolean: TRUE" << "\n";
    else
       cout << "Boolean: FALSE" << "\n";
}

/*************************************************************************/
OCLString::OCLString():OCLValue()
{
    len = 0;
    str = (char *)0;
}

OCLString::OCLString(char *s, char *c):OCLValue(c)
{
```

```
    len = strlen(s);
    str = new char[len+1];
    assert(str != 0);
    strcpy(str,s);
}


OCLValue *OCLString::evaluate()
{
    return new OCLString(this->str);
}


OCLValue *OCLString::Compare(OCLValue *e,CompType C)
{//e: Rightoperand ; this: Leftoperand
    return(e->CompareString(this,C));
}

OCLValue *OCLString::CompareString(OCLString *s,CompType C)
{//s: Leftoperand
    int t;
    bool result = false;
    //for the case when string represents a macro
    if (macrotable.look(s->str)){ //exists in macrotable
        OCLValue *treeval= macrotable.get_exprtree(s->str)->evaluate();
        int t = strcmp(treeval->Getstr(),Getstr());
    }
    else //not a macro identifier
        t = strcmp(s->Getstr(),Getstr());
    switch (C){
      case Equaltype:
          if (t==0) result = true;
          break;
      case LessThantype:
          if (t<0)  result = true;
          break;
      case GreatThantype:
          if (t>0) result = true;
          break;
    }
    if (!this->Getvarid())
```

```
        return new OCL_Boolean(result,s->Getvarid());
    else
        return new OCL_Boolean(result);
}



OCLValue *OCLString::dot(OCLValue *st)
{
//'this' ptr is an OCLString contains identifier read from input.
//if OCLString is a result of a previous dot, then get id from its
//varid,else get id from its str of 'this' ptr, look for it in vartable,
//get the iterator of the id and get the current object, apply the method
//on this object, then add id got from 'this' ptr to returned OCLValue*
//from method call.

    char *c;
    if (!(c = this->Getvarid()))
        c = this->Getstr();
    OCLValue *v = vartable.get_curobj(c)->callmethod(st->Getstr());
    v->put_varid(c);
    return v;
}

void OCLString::print_item()
{
    if (str)
        printf("String: %s\n",str);
}

void OCLString::PrintStruct(int level = 0)
{
    //Expr::PrintStruct(level);
    for(int i=0;i<level;i++)
        printf("\t");
    printf("String Expression-");
    print_item();
}

/*****************************************************************/
```

```
OCLCollection::OCLCollection(collection<anentity> *E,char *c):OCLValue(c)
{
    OCLSet_val->intersection(NULL);//ensure CLSet_val is a NULL collection
    OCLSet_val->union_col(E);      //then have OCLSet_val points to E
    set_iter = new iter_list<anentity>(OCLSet_val); //iterator on OCLSet_val
}


OCLValue *OCLCollection::evaluate()
{
    return new OCLCollection(this->OCLSet_val);
}

OCLValue *OCLCollection::Compare(OCLValue *e,CompType C)
{//e: Rightoperand  ; this: Leftoperand
    return(e->Compareset(this,C));
}

OCLValue *OCLCollection::Compareset(OCLCollection *s,CompType C)
{//s: Leftoperand
    bool result = false;

    switch(C){
      case Equaltype:
          if (s->equal(this))
             result = true;
          break;
      case LessThantype:
          if (s->lessthan(this))
             result = true;
          break;
      case GreatThantype:
          if (!(s->lessthan(this)))
             result = true;
          break;
      default:break;
    }
    if (!this->Getvarid())
        return new OCL_Boolean(result,s->Getvarid());
    else
```

```
        return new OCL_Boolean(result);

}

OCLValue *OCLCollection::plus(OCLValue *s) /* UNION operation */
{
    (this->OCLSet_val)->union_col(s->GetSetCol());
    return this;
}

OCLValue *OCLCollection::minus(OCLValue *s) /* DIFFER operation */
{
    (this->OCLSet_val)->differ(s->GetSetCol());
    return this;
}


bool OCLCollection::equal(OCLCollection *s)
{
    if ((this->OCLSet_val)->every(s->GetSetCol()) &&
        (this->Size() == s->Size()))
        return true;
    else return false;
}

bool OCLCollection::lessthan(OCLCollection *s)
{
    if ((this->OCLSet_val)->every(s->OCLSet_val) && (this->Size() < s->Size()))
        return true;
    else return false;
}

void OCLCollection::print_item()
{
    while (set_iter->POS() != NULL) { //current is not null
        set_iter->current()->print_item();
        set_iter->next();
    }
}

/*************************************************************************/
```

```
OCLSetQuant::OCLSetQuant(QuantifierType Q,int n,collection<anentity> *C,
                         char *c):OCLCollection(C,c)
{
    Qt = Q;
    num = n;
}

OCLValue *OCLSetQuant::evaluate()
{
    return new OCLSetQuant(this->Qt,this->num,this->GetSetCol());
}

OCLValue *OCLSetQuant::Compare(OCLValue *e,CompType C) // Rightoperand is the ar
{
    return(e->Comparequantset(this,C)); //Leftoperand is the arg. = this
}

OCLValue *OCLSetQuant::Comparequantset(OCLSetQuant *s,CompType C) // Leftoperand
{
    OCLValue *newset = s->evaluate();
    bool result = false;

    switch(C){
      case Equaltype:
          switch(Qt){
            case SomeType:
                if (newset->Getquant() == SomeType){
                    (newset->GetSetCol())->intersection(this->GetSetCol());
                    if (!(newset->GetSetCol()->is_empty()))
                        result = true;
                    break;
                }
                if (newset->Getquant() == EveryType) {
                    if (newset->GetSetCol()->every(this->GetSetCol()))
                        result = true;
                    break;
                }
                if (newset->Getquant() == AtleastType){
                    if (newset->GetSetCol()->atleast(newset->GetSetnum(),
                        this->GetSetCol()))
                         result = true;
```

```
                    break;
                }
                if (newset->Getquant() == AtmostType) {
                    if (newset->GetSetCol()->atmost(newset->GetSetnum(),
                        this->GetSetCol()))
                         result = true;
                    break;
                }
                if (newset->Getquant() == JustType) {
                    if (newset->GetSetCol()->just(newset->GetSetnum(),
                        this->GetSetCol()))
                         result = true;
                    break;
                }
            case AtleastType:
                if (newset->GetSetCol()->atleast(newset->GetSetnum(),
                    this->GetSetCol()))
                     result = true;
                break;
          }
      default: break;
    }

    if (!this->Getvarid())
        return new OCL_Boolean(result,newset->Getvarid());
    else
        return new OCL_Boolean(result);

}

/*************************************************************************/

OCLEntity::OCLEntity(anentity *i ,char *c):OCLValue(c)
{
    objinstance = i;
}

OCLValue *OCLEntity::evaluate()
{
    OCLEntity *ocl = new OCLEntity(objinstance);
    return ocl;
```

```
}

OCLValue *OCLEntity::dot(OCLValue *st)
{
//'this' ptr is an OCLEntity resulted from a dot expression.
//'anentity' ptr of OCLEntity actually points to a real object,
//apply the method on the current object, then add id got from 'this'
//ptr to returned OCLValue* from method call.

    OCLValue *v = get_object()->callmethod(st->Getstr());
    v->put_varid(this->Getvarid());
    return v;
}

//---------------------------- End of value.C -----------------------
```

Appendix C

Utility Code

```
//DList.h
/*****************************************************************
** Header File: DList
** Purpose: Template class for Doubly Linked lists
** Uses: NONE
** Inheritance: NONE
*****************************************************************/
#include <stdio.h>

#ifndef D_LIST
#define D_LIST

template <class T>
class DLink {
  public:
    DLink<T> *prev;
    DLink<T> *next;
    T *elemp;

    DLink(T *inelemp = NULL) { prev=NULL; next=NULL; elemp = inelemp; }
    ~DLink() { if (elemp != NULL) delete elemp;}
};

template <class T>
class DList {
  private:
    DLink<T>* first;
```

```
  DLink<T>* last;
  DLink<T>* cur;
  DLink<T>* cur2;
  DLink<T>* cur3;

public:
  DList() { first = NULL;last = NULL;cur = NULL; cur2 = NULL; }
  ~DList() {
      DLink<T>* temp;
      for(; first!=NULL; first=temp) {
          temp = first->next;
          delete first;
      }
  }

  Insert(T *elemp) {
     DLink<T> *node = new DLink<T>(elemp);

     if (first==NULL) {
          first = node;
          last = node;
          cur = node;
      }
      else {
          node->next=first;
          first->prev=node;
          first=node;
      }
  }

  Append(T* elemp) {
      DLink<T>* node = new DLink<T>(elemp);

      if (first==NULL) {
          first = node;
          last = node;
          cur = node;
      }
      else {
          last->next = node;
          node->prev = last;
```

```
        last = node;
    }
}

Append(DList<T>* listp) {
    listp->Reset();
    T* elem;
    while(elem = listp->Next())
        this->Append(elem);
}

DelCur() {
    if (cur != NULL) {
        /* Only one element on DList */
        if (first->next == NULL) {
            delete first;
            first = NULL;
            last = NULL;
            cur = NULL;
        }
        /* More than one element on DList
           cur_elem points to the first element on DList */
        else if (cur == first) {
            first = first->next;
            first->prev = NULL;
            delete cur;
            cur = first;
}
        /* More than one element on DList
           cur_elem points to the last element on DList */
        else if (cur == last) {
            last = last->prev;
            last->next = NULL;
            delete cur;
            cur = NULL;
}
        /* More than one element on DList
           cur_elem points to the middle element on DList */
        else {
            DLink<T>* temp;
            temp = cur;
```

```
            cur->prev->next = cur->next;
            cur->next->prev = cur->prev;
            cur=cur->next;
            delete temp ;
        }
     }
}

T* Prev() {
    if (cur == NULL)
        return NULL;
    DLink<T>* temp = cur;
    cur = cur->prev;
    return(temp->elemp);
}

T* Next() {
    if (cur == NULL)
        return NULL;
    DLink<T>* temp = cur;
    cur = cur->next;
    return(temp->elemp);
}

T* Next2() {
    if (cur2 == NULL)
        return NULL;
    DLink<T>* temp = cur2;
    cur2 = cur2->next;
    return(temp->elemp);
}

T* Next3() {
    if (cur3 == NULL)
        return NULL;
    DLink<T>* temp = cur3;
    cur3 = cur3->next;
    return(temp->elemp);
}

T* GoPrev() {
```

```
        cur = cur->prev;
        if (cur == NULL)
            return NULL;
        DLink<T>* temp = cur;
        return(temp->elemp);
    }

    T* GoNext() {
        cur = cur->next;
        if (cur == NULL)
            return NULL;
        DLink<T>* temp = cur;
        return(temp->elemp);
    }

    Reset() {cur = first;}
    Reset2() {cur2 = first;}
    Reset3() {cur3 = first;}
    SetDouble() {
        cur2 = cur;
    }
    T* GetCur() {
        if (cur != NULL)
            return cur->elemp;
        else
            return NULL;
    }
    int Empty() { return (first == NULL); }
};

#endif

//---------------------------- End of DList.h  -----------------------

//entity.h
#ifndef UTILITY_H
#define UTILITY_H


#define MAX 20
#include <stdio.h>
```

```
#include <stdlib.h>
#include <iostream.h>
#include <string.h>

class Value;

class anentity{
public:
    anentity(){}
    ~anentity(){}
    virtual int match(anentity*) {}
    virtual char *GetType() {}
    virtual char *get_name() {}   //for testing only
    virtual char *get_fulladdr() {} //for testing purpose only
    friend int operator==(const anentity&, const anentity&) {}
    virtual Value *callmethod(char *c) {}
    virtual void print_item() {}
};

// ================================================================

template<class Type>
class node{

private:
    Type *info;
    node<Type> *next;

public:
    node<Type>(Type* v, node<Type>* n=NULL): info(v), next(n) {}
    //default value NULL is added
    void put_next(node<Type>* n) {next = n;}
    node<Type> *get_next() {return next;}
    int search(Type* t);
    Type* get_info() {return info;}
    int match (Type* v){return (*info == *v);} //(info.isequal(&v));}     // thi
};

/* ================================================================ */

template <class Type>
```

```
class collection:public anentity{

protected:
    node<Type> *head;
    int  total;

public:
    collection() {total = 0; head = 0;}
    ~collection() {destroy();}
    int size() {return total;}
    node<Type> *ret_head() {return (node<Type>*)(void*) head;}
    virtual bool is_empty() {return total == 0 ? true : false;}

    virtual bool member(Type*);
    virtual void remove(Type*);
    virtual bool intersects(collection<Type>*);
    virtual void intersection(collection<Type>*);
    virtual bool every(collection<Type>*);
    virtual void union_col(collection<Type>*);
    virtual bool atleast(int,collection<Type>*);
    virtual bool atmost(int,collection<Type>*);
    virtual bool just(int,collection<Type>*);
    virtual void differ(collection<Type>*);

    virtual void add(Type* t);
    virtual void destroy(); // remove all items
    // int Find(const Type& t) {return head->search(t);}

    void display();
    //-----virtual functions added to accomodate derived classes ----
    virtual int  frequency(Type*){}

};

// ========================================================================
template <class Type>
class iter_list{

private:
    node<Type> *entry;
    node<Type> *pos;
```

```
public:
    iter_list(collection<Type> *lst=NULL){
        entry = lst->ret_head();
        pos = entry;
    };

    node<Type> *POS(){return pos;}
    void first(){pos = entry;}
    void next(){if (pos) pos=POS()->get_next();}
    Type *current(){
        Type* t= NULL;
        if (pos) t= POS()->get_info();
        return t;
    }

};
//=========================================================================

template <class Type>
class list : public collection<Type>{

public:

    list()  : collection<Type>() {};
    bool is_empty();
    void remove(Type* v) {del(findnb(v));}
    void del(int);
    int findnb(Type*);
    void insert(Type*, int);
    void append(Type* v) {insert(v, total+1);}
    void add(Type* v) {insert(v, total+1);}
    //should have "add" uniformly defined in derived classes
    // void modify(const Type&, int);
    int  frequency(Type*);
    Type show(int);
    list<Type>* isublist(int,int);
    list<Type>* sublist(Type* j, Type* k) {
        return isublist(findnb(j), findnb(k));
    }
};
```

```
/* ================================================================== */


template <class Type>
class bag : public collection<Type>{

public:
    bag() : collection<Type>() {};
    void add(Type*);
    int frequency(Type*);

};

/* ================================================================== */
template <class Type>
class set : public collection<Type>{

public:

    set() : collection<Type>() {}
    void add(Type* vl);
    void union_set(set<Type>*);


};

/* ================================================================== */
template<class Type>
int node<Type>::search(Type* t)
{
    node<Type>* cursor = this;
    while (cursor){
       if (cursor->match(t))
            return  1;
       else
            cursor = cursor->get_next();
    }
    return 0;

};
```

```
/* ================================================================== */

template<class Type>
bool collection<Type>::member(Type* val)
{
    if (!head)
       return false;
    node<Type> *cursor = head;
    if (cursor->search(val))
        return true;
    return false;

};

/* ================================================================== */
template <class Type>
void collection<Type>::add(Type *info)
{
    node<Type> *cursor = head, *crs = head;
    node<Type> *temp = new node<Type>(info);
    if (!head) {
        head = temp;
    }
    else
    {
        while (cursor->get_next()) {
            cursor = cursor->get_next();
        }
        cursor->put_next(temp);

    }
    total++;
};

/* ================================================================== */

template<class Type>
void collection<Type>::destroy()
{
    node<Type>* tmp;
```

```
    node<Type>* cursor;

    iter_list<Type> iter(this);
    iter.first();
    cursor = iter.POS();
    while(cursor) {

        tmp = iter.POS();
        iter.next();
        cursor = iter.POS();
        delete tmp;
    }
    cout << "SET DESTROYED!!" << endl;
    head = 0;
    total = 0;
};

/* ================================================================== */
template <class Type>
void collection<Type>::intersection(collection<Type> *lst)
{
    node<Type> *crs1 = head, *temp, *prev = head; // current one
    node<Type> *crs2 = lst->head;
    int flag = 1;
    while(crs1) {
        if (crs2->search(crs1->get_info()))
            flag = 0;
        if (flag){
            prev->put_next(crs1->get_next());
            temp = crs1;
            crs1 = crs1->get_next();
            if (temp == head)
                head = crs1;
            delete temp;
            total--;
            if (total == 0)
                head = 0;
        }
        else
        {
            prev = crs1;
```

```
            crs1 = crs1->get_next();
        }
        crs2 = lst->head;
        flag = 1;
    }
};

/* ================================================================== */

template <class Type>
bool collection<Type>::intersects(collection<Type> *lst)
{
    node<Type> *cursor1 = head;
    node<Type> *crs2 = lst->head;
    while(cursor1) {
        if (crs2->search(cursor1->get_info()))
            return true;
        cursor1 = cursor1->get_next();
        crs2 = lst->head;
    }
    return false;
};

/* ================================================================== */

template <class Type>
void collection<Type>::differ(collection<Type> *lst)
{
    node<Type> *crs1 = head, *temp, *prev = head; // current one
    node<Type> *crs2 = lst->head;
    int flag = 1;
    while(crs1) {
        if (crs2->search(crs1->get_info()))
            flag = 0;
        if (!flag){
            prev->put_next(crs1->get_next());
            temp = crs1;
            crs1 = crs1->get_next();
            if (temp == head && crs1)
                head = crs1;
            delete temp;
```

```
            total--;
            if (total == 0)
                head = 0;
        }
        else
        {
            prev = crs1;
            crs1 = crs1->get_next();
        }
        crs2 = lst->head;
        flag = 1;
    }
};
/* ========================================================================= */
template <class Type>
void collection<Type>::union_col(collection<Type> *lst)
{
    node<Type> *cursor1 = head;
    node<Type> *cursor2 = lst->head;
    while(cursor1->get_next())
        cursor1 = cursor1->get_next();
    cursor1->put_next(cursor2);
    total = total + lst->total;

};

/* ========================================================================= */

template <class Type>
bool collection<Type>::every(collection<Type> *lst)
{
    node<Type> *cursor1 = head;
    node<Type> *crs2 = lst->head;
    int flag = 1;
    while(cursor1) {
        if (crs2->search(cursor1->get_info()))
            flag = 0;
        if (flag)
            return false;
        else {
            cursor1 = cursor1->get_next();
```

```
            crs2 = lst->head;
            flag = 1;
        }
    }
    return true;
};

/* ================================================================== */

template <class Type>
bool collection<Type>::atleast(int al, collection<Type> *lst)
{

    node<Type> *cursor1 = head;
    node<Type> *crs2 = lst->head;

    int flag = 1;
    int cnt  = 0;
    while(cursor1) {
        if (crs2->search(cursor1->get_info())) {
            cnt++;
            flag = 0;
        }
        cursor1 = cursor1->get_next();
        crs2 = lst->head;
        flag = 1;
    }
    if (cnt >= al)
        return true;
    else
        return false;
};

/* ================================================================== */

template <class Type>
bool collection<Type>::atmost(int am, collection<Type> *lst)
{

    node<Type> *cursor1 = head;
    node<Type> *crs2 = lst->head;
```

```
    int flag = 1;
    int cnt  = 0;

    while(cursor1) {
        if (crs2->search(cursor1->get_info())) {
            cnt++;
            flag = 0;
        }
        cursor1 = cursor1->get_next();
        crs2 = lst->head;
        flag = 1;
    }
    if (cnt <= am)
        return true;
    else
        return false;
};


/* ========================================================================== */

template <class Type>
bool collection<Type>::just(int js, collection<Type> *lst)
{

    node<Type> *cursor1 = head;
    node<Type> *crs2 = lst->head;

    int flag = 1;
    int cnt  = 0;

    while(cursor1) {
        if (crs2->search(cursor1->get_info())) {
            cnt++;
            flag = 0;
        }

        cursor1 = cursor1->get_next();
        crs2 = lst->head;
        flag = 1;
    }
```

```
    if (cnt == js)
        return true;
    else
        return false;

};

/* ================================================================== */

template <class Type>
void collection<Type>::display()
{
    iter_list<Type> iter(this);
    iter.first();
    Type* t;

    for (; t=iter.current();iter.next()) {
        cout << "*";
        t->print(); cout << endl;
    }

};


/* ================================================================== */

/*template <class Type>
void collection<Type>::displayint()
{
    node<Type> *cursor = head;

    while (cursor) {
        cout << "*" << cursor->get_info() << endl;
        cursor = cursor->get_next();
    }
};
*/

/* ================================================================== */
```

```
template <class Type>
void collection<Type>::remove(Type* info)
{
    node<Type> *prev = head, *cursor = head, *temp;
    int flag = 1;

    while (cursor && flag) {
        if (!(cursor->search(info))){
            prev = cursor;
            cursor = cursor->get_next();
        }
        else
        {
            prev->put_next(cursor->get_next());
            temp = cursor;
            cursor = cursor->get_next();
            if (temp == head && cursor)
                head = cursor;
            delete temp;
            total--;
            if (total == 0)
                head = 0;
            flag = 0;
        }
    }
};


template<class Type>
bool list<Type>::is_empty()
{
    return total == 0 ? true : false;
};


/* ================================================================ */

template<class Type>
void list<Type>::del(int pos)
{
    node<Type> *prev, *cursor = head;
```

```
    if (pos == 1) {
        head = cursor->get_next();
        delete cursor;
        total--;
        if (total == 0)
           head = 0;
    }
    else
        if (pos > 1 && pos <= total)
        {
            for(int i = 1; (i < pos) ; i++) {
                prev = cursor;
                cursor = cursor->get_next();
            }
            prev->put_next(cursor->get_next());
            delete cursor;
            total--;
        }
};

/* ========================================================================== */

template<class Type>
void list<Type>::insert(Type* val, int pos)
{
    node<Type> *prev, *cursor = head, *temp = new node<Type>(val);
    int i;
    if (!head) {
        head = temp;
        total++;
    }
    else
        if (pos == 1) { // insert at head
            temp->put_next(head);
            head = temp;
            total++;
        }
        else
            if (pos > 1)
            {
```

```
                for(i = 1; (i < pos) && cursor->get_next(); i++) {
                    prev = cursor;
                    cursor = cursor->get_next();
                }
                if (!cursor->get_next() && (i < pos))
                    cursor->put_next(temp);
                else {
                    temp->put_next(cursor);
                    prev->put_next(temp);
                }
                total++;
            }
};

/* ========================================================================= */

template<class Type>
int list<Type>::findnb(Type* val)
{
    int cnt = 0;
    if (!head)
        return cnt;
    node<Type> *cursor = head;
    while(cursor) {
        cnt++;
        if (cursor->match(val))
             return cnt;
        cursor = cursor->get_next();
    }
    return 0;
};

/* ========================================================================= */

template<class Type>
int list<Type>::frequency(Type* val)
{
    node<Type> *cursor = head;
    int count = 0;
    if (!head)
        cout << "List is empty!" << endl;
```

```
    else {
        while (cursor) {
            if (cursor->match(val))
                count++;
            cursor = cursor->get_next();
        }
    }
    return count;
};

/* ================================================================== */

template<class Type>
Type list<Type>::show(int pos)
{
    node<Type> *cursor = head;
    if ((pos <= total) && (cursor))
    {
        for (int i=1; i < pos;i++)
            cursor = cursor->get_next();
        return(cursor->get_info());
    }
    return 0;
};

/* ================================================================== */


template<class Type>
list<Type>* list<Type>::isublist(int min, int max)
{
    list<Type> *lst;
    lst = new list<Type>;
    node<Type> *cursor = head;
    if ((!head) || (min > max))
        return lst;
    for (int i=1; (i <= max) && cursor; i++){
        if ((i >= min) && (i <= max))
            lst->append(cursor->get_info());
        cursor = cursor->get_next();
    }
```

```
    return lst;
};

/* ================================================================== */
template <class Type>
void set<Type>::union_set(set<Type> *st)
{
    node<Type> *cursor1 = head, *prev = head;
    node<Type> *crs2 = st->head;
    int flag = 1;

    while(crs2) {
       while(cursor1 && flag) {
            if (cursor1->match(crs2->get_info()))
                flag = 0;
            else
                prev = cursor1;
            cursor1 = cursor1->get_next();
        }
        if (flag) {
            prev->put_next(crs2);
            total++;
        }
        cursor1 = head;
        prev = head;
        crs2 = crs2->get_next();
        flag = 1;
    }

};


/* ================================================================== */

template <class Type>
void bag<Type>::add(Type *info)
{
    node<Type> *cursor = head, *crs = head;
    node<Type> *temp = new node<Type>(info);

    if (!head) {
```

```
        head = temp;
    }
    else
    {
        while (cursor->get_next()) {
            cursor = cursor->get_next();
        }
        cursor->put_next(temp);
    }
    total++;
};

/* ========================================================================= */

template <class Type>
int bag<Type>::frequency(Type* info)
{
    node<Type> *cursor = head;
    int count = 0;

    if (!head)
        cout << "List is empty!" << endl;
    else
    {
        while (cursor)
        {
            if (cursor->match(info))
                count++;
            cursor = cursor->get_next();
        }
    }
    return count;
};

/* ========================================================================= */

template <class Type>
void set<Type>::add(Type* vl)
{
    node<Type>* cursor = head;
```

```
    if (!cursor->search(vl))
    {
        head = new node<Type>(vl,head);
        total++;
    }
};

#endif

//---------------------------- End of entity.h  ------------------------
```

Appendix D

Grammar

```
%{
#include <stdio.h>
#include "prog.h"
#include "expr.h"
#include "qual.h"
#include "DList.h"

#define YYERROR_VERBOSE
#define YYDEBUG 1

char error_message[256];
int char_count;
int line_count;
Prog *program;

int yylex(void);
int yyerror(char*);

%}

%union
{
        char                    instring[64];
        int                     innumb;
        int                     CollectionType;
        Prog                    *prog_ptr;
        DList<Expr>             *expr_list_ptr;
```

```
        Expr                    *expr_ptr;
        Qualifier               *qual_ptr;
        DList<Qualifier>        *qual_list_ptr;
        Generator               *generator_ptr;
        Localdef                *localdef_ptr;
        Hasclassqual            *hasclass_ptr;
        Hasclasswithqual        *hasclasswith_ptr;
}

%type   <prog_ptr>              program
%type   <expr_list_ptr>         expr_list
%type   <expr_ptr>              expr
%type   <qual_list_ptr>         qualifier_list
%type   <qual_list_ptr>         quals
%type   <qual_ptr>              qualifier
%type   <generator_ptr>         generator
%type   <localdef_ptr>          local_def
%type   <hasclass_ptr>          hasclass_qual
%type   <hasclasswith_ptr>      hasclasswith_qual
%type   <CollectionType>        collection_kind


%token  <instring>      COMMATK
%token  <instring>      SEMICOLONTK
%token  <instring>      PLUSTK
%token  <instring>      MINUSTK
%token  <instring>      MULTTK
%token  <instring>      DIVTK
%token  <instring>      NOTTK
%token  <instring>      LIMPLTK
%token  <instring>      IDENTIFIERTK
%token  <innumb>        CONSTANTTK
%token  <instring>      SIZETK
%token  <instring>      SETTK
%token  <instring>      LISTTK
%token  <instring>      BAGTK
%token  <instring>      SOMETK
%token  <instring>      ATLEASTTK
%token  <instring>      JUSTTK
%token  <instring>      ATMOSTTK
%token  <instring>      EVERYTK
```

```
%token  <instring>      OFTK
%token  <instring>      ASTK
%token  <instring>      LITERALTK
%token  <instring>      HASCLASSTK
%token  <instring>      WITHTK
%token  <instring>      ANDTK
%token  <instring>      ORTK
%token  <instring>      EQTK
%token  <instring>      GTTK
%token  <instring>      LTTK
%token  <instring>      NOTEQTK
%token  <instring>      GTOREQTK
%token  <instring>      LTOREQTK
%token  <instring>      DEQTK
%token  <instring>      NOTDEQTK
%token  <instring>      THREEDOTTK
%token  <instring>      DOTTK
%token  <instring>      DIFFERTK
%token  <instring>      UNIONTK
%token  <instring>      LSQBKTK
%token  <instring>      RSQBKTK
%token  <instring>      LCBKTK
%token  <instring>      RCBKTK
%token  <instring>      LBKTK
%token  <instring>      RBKTK
%token  <instring>      BARTK


%nonassoc    HASCLASSTK WITHTK
%left        ANDTK   ORTK
%nonassoc    UNOT
%nonassoc    EQTK GTTK LTTK  NOTEQTK GTOREQTK LTOREQTK DEQTK NOTDEQTK

%nonassoc    THREEDOTTK
%left        MINUSTK  PLUSTK
%left        MULTTK  DIVTK
%nonassoc    UMINUS
%left        DOTTK

%nonassoc    UQUANT
```

```
%left        DIFFERTK
%left        UNIONTK

/* Grammar follows */

%%

program:
        expr_list
        {
                yydebug = 1;
                $$ = new Prog($1);
                program = $$;
                $$->PrintStruct();
        }
;

expr_list:
        expr
        {
                $$ = new DList<Expr>;
                $$->Insert($1);
        }|
        expr COMMATK expr_list
        {
                $$ = $3;
                $$->Insert($1);
        }
;

expr:
        expr MULTTK expr
        {
                $$ = new Expr($1,$3,MultOpcode);
        }|
        expr DIVTK expr
        {
                $$ = new Expr($1,$3,DivOpcode);
        }|
        expr PLUSTK expr
        {
```

```
        $$ = new Expr($1,$3,PlusOpcode);
}|
expr MINUSTK expr
{
        $$ = new Expr($1,$3,MinusOpcode);
}|
SIZETK expr   %prec UMINUS
{
        $$ = new Expr($2,NULL,SizeOpcode);
}|
expr ANDTK expr /* MultOpcode is *, and operations */
{
        $$ = new Expr($1,$3,MultOpcode);
}|
expr ORTK expr /* PlusOpcdoe is +, or, union operations */
{
        $$ = new Expr($1,$3,PlusOpcode);
}|
NOTTK expr       %prec UNOT
{
        $$ = new Expr($2,NULL,NotOpcode);
}|
expr EQTK expr
{
        $$ = new Expr($1,$3,EqOpcode);
}|
expr GTTK expr
{
        $$ = new Expr($1,$3,GtOpcode);
}|
expr LTTK expr
{
        $$ = new Expr($1,$3,LtOpcode);
}|
expr GTOREQTK expr
{
        $$ = new Expr($1,$3,GtEqOpcode);
}|
expr LTOREQTK expr
{
        $$ = new Expr($1,$3,LtEqOpcode);
```

```
}|
expr NOTEQTK expr
{
        $$ = new Expr($1,$3,NotEqOpcode);
}|
expr UNIONTK expr /* PlusOpcode is+, union, or operations */
{
        $$ = new Expr($1,$3,PlusOpcode);
}|
expr DIFFERTK expr /* MinusOpcode is -, differ operations */
{
        $$ = new Expr($1,$3,MinusOpcode);
}|
SOMETK  OFTK expr       %prec UQUANT
{
        $$ = new Expr($3,NULL,SomeOpcode);
}|
EVERYTK  OFTK expr      %prec UQUANT
{
        $$ = new Expr($3,NULL,EveryOpcode);
}|
ATLEASTTK expr OFTK expr   %prec UQUANT
{
        $$ = new Expr($2,$4,AtleastOpcode);
}|
JUSTTK expr OFTK expr %prec UQUANT
{
        $$ = new Expr($2,$4,JustOpcode);
}|
ATMOSTTK expr OFTK expr %prec UQUANT
{
        $$ = new Expr($2,$4,AtmostOpcode);
}|
expr DOTTK expr
{
        $$ = new Expr($1,$3,DotOpcode);
}|
expr DOTTK LSQBKTK expr RSQBKTK
{
        $$ = new Expr($1,$4,DotSBKOpcode);
}|
```

```
        LBKTK expr RBKTK
        {
                $$ = $2;
        }|
        IDENTIFIERTK
        {
                $$ = new Expr_string($1); //new OCLString($1);
        }|
        CONSTANTTK
        {
                $$ = new Expr_num($1); //new Number($1);
        }|
        LITERALTK
        {
                $$ = new Expr_string($1); //new OCLString($1);
        }|
        IDENTIFIERTK LBKTK expr_list RBKTK
        {
                $$ = new Expr_methodcall($3,$1);
        }|
        collection_kind LSQBKTK qualifier_list BARTK expr RSQBKTK
        {
                $$ = new Expr_query($3,$1,$5);
        }|
        collection_kind LCBKTK expr_list RCBKTK
        {
                $$ = new Expr_special($3,$1);
        }|
        collection_kind LCBKTK expr THREEDOTTK expr RCBKTK
        {
                $$ = new Expr_subrange($1,$3,$5);
        }
;

collection_kind:
        SETTK
        {
                $$ = 1;
        }|
        LISTTK
        {
```

```
                $$ = 2;
        }|
        BAGTK
        {
                $$ = 3;
        }
;

qualifier_list: /* empty */
        {
                $$ = NULL;
        }|
        quals
        {
                $$ = $1;
        }
;

quals:
        qualifier
        {
                $$ = new DList<Qualifier>;
                $$->Insert($1);
        }|
        qualifier SEMICOLONTK quals
        {
                $$ = $3;
                $$->Insert($1);
        }
;

qualifier:
        generator
        {
                $$ = $1;
        }|
        local_def
        {
                $$ = $1;
        }|
        hasclass_qual
```

```
        {
                $$ = $1;
        }|
        hasclasswith_qual
        {
                $$ = $1;
        }|
        expr
        {
                $$ = $1;
        }
;

generator:
        IDENTIFIERTK LIMPLTK expr
        {
                $$ = new Generator($1,$3);
        }
;

local_def:
        IDENTIFIERTK ASTK expr
        {
                $$ = new Localdef($1,$3);
        }
;

hasclass_qual:
        expr HASCLASSTK expr
        {
               $$ = new Hasclassqual($1,$3);
        }
;

hasclasswith_qual:
        expr HASCLASSTK expr WITHTK expr
        {
              $$ = new Hasclasswithqual($5,$1,$3);
        }
;
```

```
%%

int yyerror(char* message)
{
        printf("%s\n(%d,%d)",message,line_count,char_count);
        exit(0);
}
```